Graphical Development Interface and Stream Analyzer for Apache Spark

By

Arun Sharma

A project report submitted in the partial fulfillment of the requirements for the degree of

Masters of Engineering

in

Computer Engineering

University of Puerto Rico

Mayaguez Campus 2018

Approved By:

Emmanuel Arzuaga, Ph. D Professor, Graduate Committee	Date
Pedro Rivera, Ph. D Professor, Graduate Committee	Date
Manuel Rodriguez, Ph. D President, Graduate Committee	Date
Rocío Zapata, M.A Graduate School Representative	Date
Dr. José Colom Ustariz Director, ECE DEpartment	Date

ABSTRACT ENGLISH

Technology advances and doubles itself every two years. According to Moore's law [1], number of transistors on an integrated circuit design, doubles every two years. With this increase, increases the processing power of the devices. But the hurdle comes in when we human beings are not able to find apt solutions to fully utilize this number-crunching power of the devices. To solve this big problem, the solution needs to be big as well. Throughout the history of computing devices, researchers and programmers worldwide have been trying to solve this issue by making computing devices work together in a networked fashion, and this has been termed as **Distributed Processing**.

Apache ™ a non-profit corporation, that develops and distributes free and open source frameworks, tools and SDKs, has been constantly trying to come up with a better solution to help achieve maximum processing speeds over a cluster of computing devices interconnected on a network. There are many widely used tools such as Apache Spark, Apache Storm, Apache Flink, Twitter's Heron, Alluxion Open Foundation's "Alluxion" framework. All these tools help in solving big data processing problems. Based on these tools, an analyst with prior knowledge of programming, can analyze huge database sets consisting of millions and billions of data rows. But it needs hands on knowledge of programming and most of the times very deep understanding of big data processing algorithms. In this work, we developed a graphical development interface (GDI) that is aimed to minimize the effort required to use these available tools when analyzing big data. We name the system as "**Stream Analyzer**". But more than just minimizing that effort for the typical user, this system will also allow these analysis tools to be used by individuals with little, or even none, experience in software development.

Stream Analyzer is a collection of tools that allows a novice user to quickly setup a stream processing environment. It is a system based on plugins/components which can be dragged dropped on to the topology design area and can be interconnected. These components serve as basis for doing the main processing work behind the scenes. No prior knowledge of programming is required as the plugins/components can be installed from the plugin management area. It is a multiuser system with ability to provide topology and project management. Furthermore, at the end of the processing end point components, user can attach a report generation plugin, provide configuration parameters and finally run the topology to actually see the live report generation in the report view area of the GDI. The core idea behind this GDI is to let any user quickly learn the tools and use them to analyze the big data present in the huge world of internet without any need to code.

ABSTRACT SPANISH

La tecnología avanza y se duplica cada dos años. De acuerdo con la ley de Moore [1], el número de transistores en un diseño de circuito integrado se duplica cada dos años. Con este aumento, aumenta la potencia de procesamiento de los dispositivos. Pero el obstáculo se produce cuando los seres humanos no somos capaces de encontrar soluciones aptas para utilizar por completo este poder numérico de los dispositivos. Para resolver este gran problema, la solución también debe ser grande. A lo largo de la historia de los dispositivos informáticos, los investigadores y programadores de todo el mundo han estado tratando de resolver este problema haciendo que los dispositivos informáticos trabajen juntos en una red, y esto se ha denominado procesamiento distribuido.

Apache TM, una corporación sin fines de lucro, que desarrolla y distribuye frameworks, herramientas y SDK de código abierto y gratuito, ha estado constantemente tratando de encontrar una mejor solución para ayudar a alcanzar velocidades de procesamiento máximas sobre un grupo de dispositivos informáticos interconectados en una red. Hay muchas herramientas ampliamente utilizadas como Apache Spark, Apache Storm, Apache Flink, Twitter's Heron, Alluxion Open Foundation, el marco "Alluxion". Todas estas herramientas ayudan a resolver problemas de procesamiento de big data. Con base en estas herramientas, un analista con conocimiento previo de programación puede analizar enormes conjuntos de bases de datos que consisten en millones y miles de millones de filas de datos. Pero necesita conocimientos prácticos de programación y la mayoría de las veces una comprensión muy profunda de los algoritmos de procesamiento de big data. Para aliviar el dolor de las personas que desean analizar los datos en la mano y tomar

decisiones basadas en la salida sin tener una experiencia de programación, desarrollamos una interfaz gráfica de desarrollo (GDI) denominada "Stream Analyzer".

Stream Analyzer es una colección de herramientas que permite a un usuario novato configurar rápidamente un entorno de procesamiento de flujo. Es un sistema basado en complementos / componentes que se puede arrastrar al área de diseño de topología y se puede interconectar. Estos componentes sirven de base para realizar el trabajo principal de procesamiento detrás de escena. No se requiere ningún conocimiento previo de programación ya que los complementos / componentes se pueden instalar desde el área de administración de complementos. Es un sistema multiusuario con capacidad para proporcionar topología y gestión de proyectos. Además, al final de los componentes del punto final de procesamiento, el usuario puede adjuntar un plugin de generación de informes, proporcionar parámetros de configuración y finalmente ejecutar la topología para ver realmente la generación de informes en vivo en el área de visualización de informes del GDI. La idea central detrás de este GDI es permitir que cualquier usuario aprenda rápidamente las herramientas y las use para analizar los grandes datos presentes en el enorme mundo de internet sin necesidad de codificar.

“*I dedicate this project to my family and friends, for the support they gave me each day to accomplish the objectives that I propose in life.*”

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

Data is divided into two main categories, structured, which is the processed data and ready for rigorous analysis and unstructured data, which is the raw data generated everyday throughout the world by millions of computer programs. More than 80% of data is unstructured according to a publication by IBM Company [12]. This unstructured data is not very much valuable until we can find some insights from it that can actually help take better data based decisions for any organization. To make it valuable, data is processed and converted into a form that can easily be visualized and understood by managing staff of a company or organization. The data to be processed is generally millions or even billions of unstructured data rows, and for this huge amount of data to be processed, organizations need special infrastructure which consists of network connected computers, storage systems and software processing frameworks. Apache Spark is one such distributed data processing framework that helps speedup the process of distributed data processing by using a computer's main memory as a temporary storage solution while the data is being processed. Main memory is very much faster compared to hard disks and this speeds up the processing by 10~100x faster when Apache Spark is used as a processing framework [3].

Apache Spark is a framework that helps distributed processing application developers develop executable software jobs that can processes millions to billions of data rows in a really fast manner over a network of connected nodes in a master-slave fashion. But, the main issue with developing jobs for such framework is that it increases the overall headache of writing the same logical code again and again for those parts of jobs which are generally used in every other job. For example, a map-reduce task such as filtering or aggregating is used many times while developing an Apache Spark map reduce job. To help developers to quickly implement their ideas instead of dedicating more time than necessary on rewriting the same code again and again, we developed a graphical development framework(GDI) called "*Stream Analyzer*". Stream Analyzer is a web based framework which helps in developing distributed processing jobs in a very easy yet efficient manner. Developers can use the GDI, to drag and drop the already implemented code in the form of components or plugins to the design area where a job's topology is designed. These components contain those parts of code which generally need to be re-written when hand-coding a map-reduce job for Apache Spark. This way, developers can just focus on designing the topology for the job. Once done, the code is automatically converted into a deployable Apache Spark job by just one click of a button on the GDI area.

1.1 Objectives

1. The main objective of this project was to develop general purpose data analysis web based IDE with graphical programming and development interface. To achieve this big objective, we had to make sure that our project can be extended in terms of functionality by the means of plugin system. The system should allow end users to develop and reuse the components or plugin that they develop.

2. *Cross Platform*, the system should be easy to setup and use on any platform. As the system uses JVM and NODE.JS as backend technologies, they make the system pretty much cross platform itself. Even the applications/jobs developed using the system are basically well organized JAVA projects which can be shared with others.

3. *Easily Accessible GUI*, we wanted to make the GUI easily accessible from anywhere around the world. To meet this objective, we had to develop the web based GUI which is accessible through any modern web browser such as Google Chrome or Mozilla Firefox.

4. *Divide the workspace per user*, the main aim was to let everyone use the system instead of having it installed/setup over each user machine. We introduced authentication mechanism to let the users sign up and login to their personal workspace. This way the whole system can be setup and offered as a SAAS system.

5. *Open-source and easily distributable*, we made the system open sourced which allows more incoming contributors to help extend the system more by developing plugins. These plugins can be made open sourced as well or can be proprietary depending on developer's choice.

1.2 Contributions of the project

The main contributions of this distributed processing framework GDI are:

1. Provide an easily accessible interface to beginner developers in the field of distributed processing applications development and especially for Apache Spark. The interface can be used by anyone to quickly and easily develop deployable Apache Spark jobs without any need of getting involved in writing the code.

2. As we have made this software open-sourced, there is vast possibility of wide extension of this software and be accessible to every developer out there, for free.

3. It is an all in one framework which provides a complete package of designing, deploying/running and generating an analysis report of the job under process. This way there is no need for external third party tools to perform chunks of the task.

1.3 Document organization

The remaining of the document is organized in following way:

1. Chapter 2, Literature Review: In this chapter, we discuss about the background of the tools and frameworks that are used in this project. It also informs about different frameworks that were implemented in the past, and are similar to what we developed.

2. Chapter 3, Architecture: Discusses about the main architecture & design of "*Stream Analyzer*" software. It describes the main modules/parts in which the whole software is divided and how they interact with each other.

3. Chapter 4, System Implementation: Here we discuss about the implementation methods and materials that we used to develop the "*Stream Analyzer*" software. We get into the details of the logical code that is responsible for making the software work and also the result achieved after running the software.

4. Chapter 5, Conclusion & Future Work: In this chapter we discuss the conclusion we made after developing and successfully executing the software multiple times and also discuss its limitations and future scope.

2. LITRATURE REVIEW

2.1 Distributed Processing System

A distributed processing system consists of multiple computing machines connected to each other over a network, which communicate with each other to carry out a common task by processing different parts of that task in parallel. There is a management node or leader node that leads all other slave or worker nodes into performing processing intensive tasks to achieve a common goal.

Each node may also be used as a data storage node (Fig. 1). The data among such nodes is kept in sync with the master node. All these data nodes are accessible to other nodes in the cluster through a global endpoint. A client generally interacts directly with the master node to perform basic tasks such as deploying a job, modifying the cluster and monitoring the system.

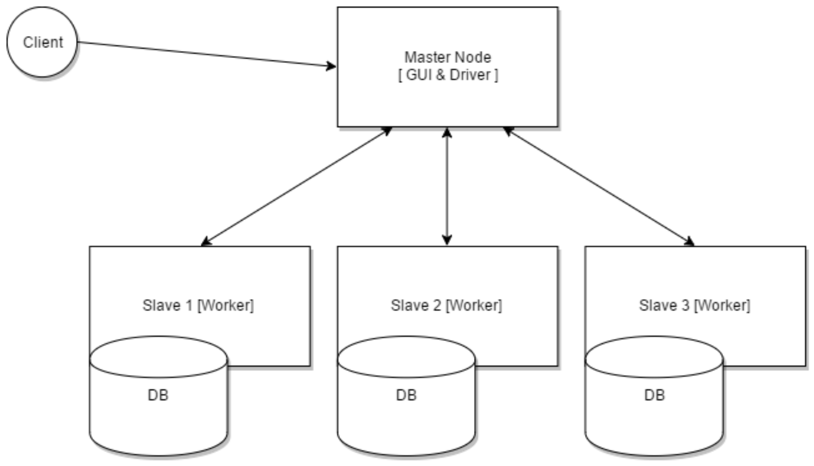


Fig. 1 Distributed Processing System

A job received by the master node is distributed to the worker nodes for processing. As the big data is generally millions or billions of records in count, reads-writes are performed in coarse grained manner for such large datasets.

2.2 Big Data and Map Reduce Paradigm

Big Data is a term given to the datasets with record count in millions to billions of number of rows (and even more!). It is a challenging task to perform faster yet reliable computations over such large amount of data even with today's large scale processing hardware. That is why developers use new and efficient programming paradigms to tackle the challenge of processing and analyzing big data. Generally used paradigm for performing such data intensive operations is one called Map Reduce.

Map Reduce was first used in LISP programming language [4]. It is basically divided in two parts which, as the name suggests, are Map and Reduce. The Map function is used to map the user specified input data into partially categorized or aggregated form. Further, Reduce function reduces or groups the mapped data into the final output as shown in the figure (Fig. 2). Map Reduce paradigm allows for parallel processing of data over a cluster of machines. It also hides a lot of complexity of processing from the user and provides methods to implement only the required logic at application level of programming.

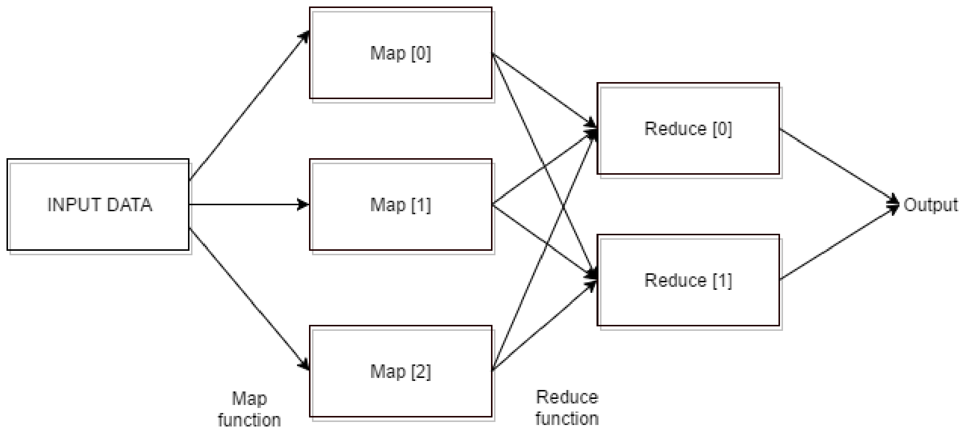


Fig. 2 Map-Reduce method of processing Big Data.

2.3 Hadoop and Spark

Generally, basic tasks performed over big data include mapping, filtering, aggregating, sorting, counting, reducing and reporting [9]. Some of the examples of tools to perform all such tasks are Apache Hadoop tools [2], Storm and Heron by Apache and Apache Flink. All these tools are frameworks that leverage the processing power of clustered and distributed processing environment to perform the basic data analysis tasks at fast speeds. Apache Hadoop is an ecosystem that consists of many helpful tools and SDKs to load, store, transform and query the terabytes of data in a quick, secure, consistent and fault-recoverable method.

Apache's Hadoop is a collection of software tools that allow for distributed processing of large datasets across a cluster of connected computing devices [2]. Doug Cutting and Mike Cafarella developed Hadoop at the University of Michigan [4]. It is based on Map Reduce architecture implemented by Google. It later became part of Apache foundations open source family and since then has been called Apache Hadoop Framework.

The problem with Hadoop ecosystem is that it is vast and very modularized that makes it complex to manage large data processing jobs. Furthermore, if not managed properly, the jobs deployed over Hadoop ecosystem may become slow and sluggish. To solve this problem, Apache came up with a new framework called Spark. Jobs developed with Spark's SDK can achieve processing speed of up to 100 times faster in memory and 10 times faster on disk compared to map-reduce jobs developed on core Hadoop system [3]. The main reason behind this considerable speedup of jobs is that Spark uses the main memory to store the data under processing. Apache Spark lets users persist the data into main memory which basically serves as a cache for the cluster of machines. Each machine communicates with other on a network through Spark system and data can be shared at very high speeds because the system is not dependent on disks to perform data intensive operations.

Although Apache Spark runs on Hadoop, It uses only the parts of the ecosystem. Spark doesn't have its own storage management system, hence it relies on Hadoop's HDFS for storing and loading data over a Hadoop cluster. In addition, Spark uses Hadoop's YARN to work in cluster mode and to manage the resource allocations per node in the cluster.

2.4 Cluster Computing with Spark

Parallel data processing has been effectively and efficiently achieved by Apache Spark framework in a clustered environment made up of commodity hardware. Spark is a better version of Map Reduce paradigm because it exposes the high speed I/O power of main memory to Map Reduce jobs. Jobs executed on Apache Spark frameworks execution engine, have access to read and write methods on the main memory of each slave node in the clustered environment. Spark makes the border between the main memories of each machine in the distributed processing environment seamless. This way the read write times for a job reduce drastically and make the job faster compared to those executed on Hadoop ecosystem alone.

Apache Spark also targets some unconventional application architectures where each chunk of code getting executed across machines in a cluster, wants to reuse the processed data sets from previous stages of execution [5]. This architecture doesn't follow the acyclic data flow graphs paradigm. More specifically, Spark targets two types of such applications which are [5]:

1. Iterative Jobs: Such as machine learning applications which include learning from reusing resulting datasets from previous stages.
2. Interactive Analytics: These jobs allow for interactive querying over datasets under processing.

2.5 HDFS

Big data processing jobs which are executed over Apache Hadoop's cluster with disk I/O involvement are generally slow compared to Spark because spark uses the main memory as its default storage method. Apache Spark doesn't provide its own disk storage mechanism, rather it relies on other clustered storage management solutions. Apache Hadoop's HDFS is one such distributed storage mechanism with the help of which, Spark jobs can easily flush or read data to or from the disk storage distributed among a number or all of the nodes in a clustered environment. HDFS stands for Hadoop Distributed Files System and is like a UNIX like file storage system but lacks a lot of standards so as to enhance the overall performance of jobs under execution. HDFS saves the information about the application data separate from the application data itself in nodes which are termed as NAMENODES. The application data is stored in other nodes which are termed as DATANODES [6].

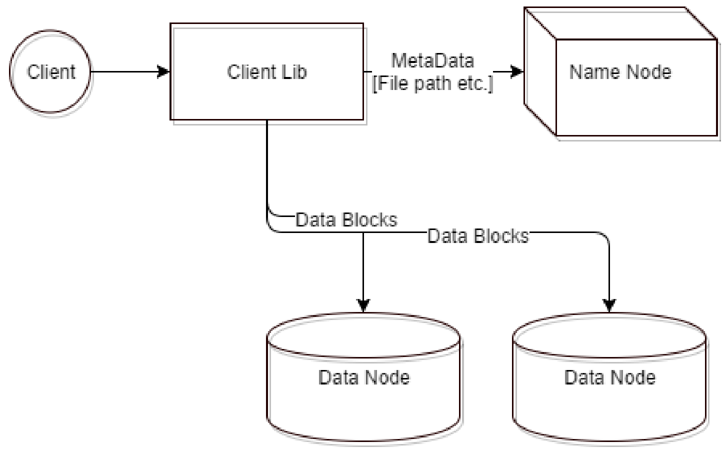


Fig. 3 HDFS Client library

HDFS also provides a client library (Fig3.) which serves as an interface between the client applications or jobs and the distributed file storage system that HDFS provides as a service. The client library tells the name-nodes the path to the file it wants to store in the data-nodes. Further, the client transfers each block to the data-nodes and eventually informs the name-nodes of data creation.

2.6 Spark RDDs

Spark RDD or Resilient Distributed Dataset is an abstraction over distributed file systems that allows spark engine based applications to have unified access to the underlying storage. Hadoop based applications follow acyclic data flow graphs but there are more types of applications emerging because of the ever increasing big data which need to reuse the datasets while executing. The Spark research and development team at the University of California, Berkeley [7], came up with RDDs to tackle this problem. Spark uses RDDs to store data in the main memory across the cluster machines. Spark engine allows these RDDs to be cached and reused by an application under execution. In addition, these datasets can also be flushed to Hadoop file system called HDFS by a method called check-pointing.

Spark RDDs provide a fault recoverable method for storing data in Hadoop file system. Each RDD has enough information with it about how it was generated based on the input data and same information is used when a fault occurs to reconstruct itself by the RDD. This way the dataset under fault, can be reconstructed and the application can easily recover from a failure.

2.7 Spark D-Streams

Latest hardware technologies are providing never before attained number crunching power to the processing units in current world's commodity hardware. Faster speeds of data processing also generate large data sets. Analyzing these datasets in real time has become a very important task for the data analysts of current competitive market.

Spark D-Streams or Discretized streams is a mechanism develop at the University of Berkeley [8] for processing the data that arrives in real time. D-Streams treat streaming computations as a series of deterministic batch computations divided in small time intervals [8]. D-Streams were developed to alleviate the problems that exist with conventional stream processing methods where data arrived was stored and processed in per record manner. Processing data per record provides data inconsistency and is also a fault intolerant method. D-Streams divide the incoming data chunks into time intervals which means, the data arrived at a particular instant belongs to that instant or time interval. Each interval batch of data is then fed to Map Reduce functions for processing further. This method provides data consistency as the data is batched into intervals and same data is available to all the nodes in the cluster. Furthermore, it is a fault tolerant method as the data being processed is processed atomically i.e. at that particular instant when it was received.

2.8 Related Work

Many big data processing frameworks and SDKs are available online which are also free and open sourced. Some of these frameworks are specifically developed to perform big data stream processing and many of these tools outperform others. Some of the tools that are related to our project include:

1) Talend Studio

   It is a feature rich solution for performing mundane to enterprise level big data processing tasks. Talend Studio is a free and open sourced application software that allows data extraction, transformation and integration tasks in a clustered environment of interconnected commodity computing devices [10].

2) Pentaho

   Pentaho is a Hitachi Group Company [11] which provides big data integration, analytics and business analytics as service to its clients worldwide. The company provides a set of tools to these big data processing tasks on a variety of operating system flavors.

These SDKs provide all the basic necessary features for handling big data processing and analysis. But the main problem with these tools is that these tools are standalone applications which cannot be accessed online as a service. That is the main reason which led us to develop a new framework named Stream Analyzer which provides a graphical programming interface which is accessible through a web browser.

3. ARCHITECTURE

The system is divided in three major parts (as shown in figure 4 below) each of which is needed for the system to function fully and properly.

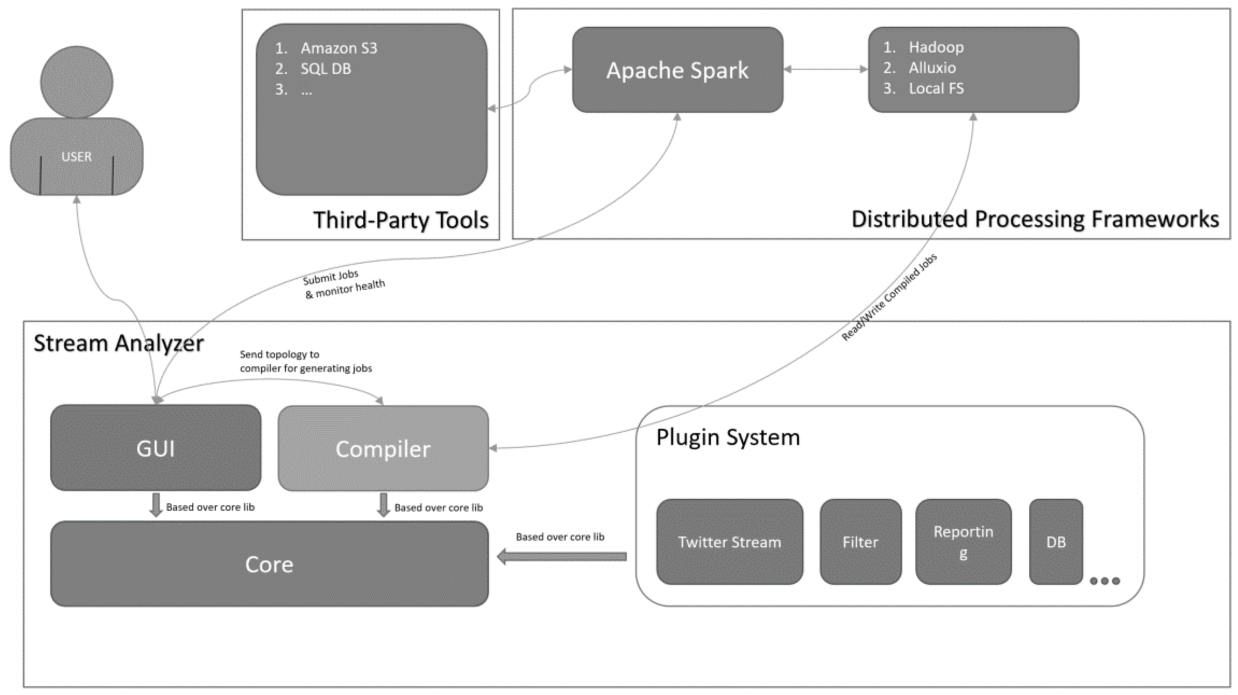


Fig. 4 Stream Analyzer Architecture

3.1 Stream Analyzer

It consists of four major modules which help the end user design the topology and compile it into a java archive file which is a representation of a job that is understandable and executable by Apache Spark engine. The above mentioned modules are:

3.1.1 Web based GUI

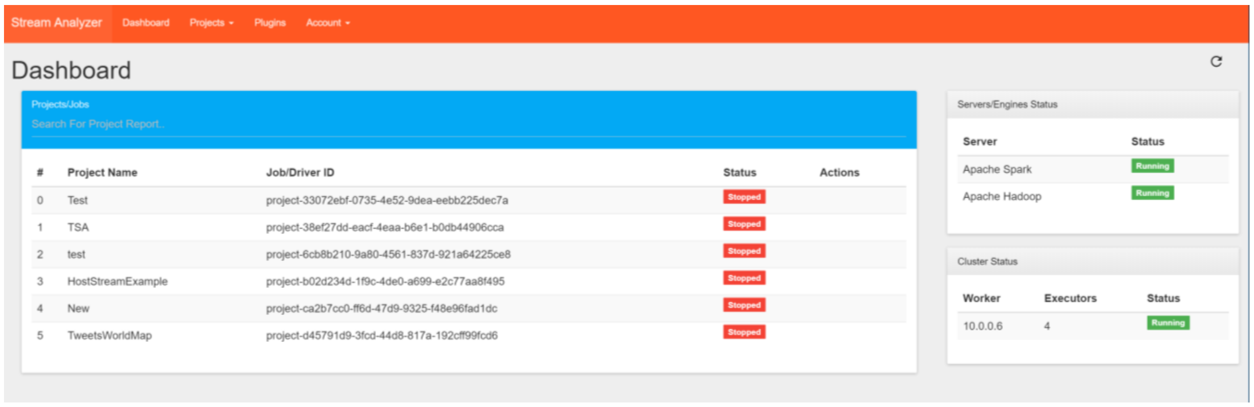
The main point of interaction between the end user and the system is a web based GUI that can be accessed over local networks or/and the Internet. Users can register and login to the system.

Each user has his/her own workspace in the system and can design, compile and run the project topologies totally isolated from other users' workspace.

When a user is successfully logged in to the system, he/she is redirected to the dashboard screen (as shown in figure 5 below). Dashboard gives an overview of what is happening in the overall system. Users can quickly search for a job and see its status and can also keep a watch over the live report corresponding to that particular job. The status of a job can either be "*Running*" or "*Stopped*" as per the standard life cycle statuses of a job provided by Apache Spark engine.

The dashboard also keeps the user updated about the status of backend engines. Apache Spark and Hadoop are the two base engines for this system. The status of the engines can be either "*Running*" or "*Stopped*". The system needs both the engines to be already running to function properly, hence, the engines need to be started before the deployment of any job. Engines can easily be started by running a shell script named "*start_services.sh*" from the root terminal by the system administrator. The script is present in the installation directory of the system (Stream Analyzer).

Furthermore, the dashboard also shows the status of worker nodes connected to the Apache Spark master node. For heavy jobs to function properly, multiple workers are needed which can quickly distribute and process the incoming stream of data. Again, the worker can be under one of the two states which are "*Running*" or "*Stopped*". A disconnected node is also represented as "*Stopped*" on the dashboard screen.



#	Project Name	Job/Driver ID	Status	Actions
0	Test	project-33072ebf-0735-4e52-9dea-eebb225dec7a	Stopped	
1	TSA	project-38ef27dd-eacf-4eaa-b6e1-b0db44906cca	Stopped	
2	test	project-6cb8b210-9a80-4561-837d-921a64225ce8	Stopped	
3	HostStreamExample	project-b02d234d-1f9c-4de0-a699-e2c77aa8f495	Stopped	
4	New	project-ca2b7cc0-ff6d-47d9-9325-f48e96fad1dc	Stopped	
5	TweetsWorldMap	project-d45791d9-3fcd-44d8-817a-192cff99fcd6	Stopped	

Server	Status
Apache Spark	Running
Apache Hadoop	Running

Worker	Executors	Status
10.0.0.6	4	Running

Fig. 5 GUI: Dashboard

A project is a collection of a job topology, compiled job and the metadata to tell the system about how to manage the project.

3.1.2 Core Module

The topology created by end user in the topology design area of the web based GUI, is converted to a serialized object in the form of JSON and is sent to the backend server where the GUI code is executed. Backend server has API endpoints that listen for the request to compile the topology into a java archived job i.e. a JAR file. This jar file contains everything needed by the Apache Spark engine to process the topology designed by the end user. One of the important modules packed into the jar file is the core module. It is not directly embedded into the jar but is used by the plugins developers to develop the components visible to the end user on web GUI.

There are some standards to be followed by the plugin developers so as to have their plugins work in error free manner. Each component on the web GUI represents a stage of processing. Each stage type extends directly from the base stage type and implements IConnector interface (as shown in figure 6 below). The interface makes each stage connectable to other in the topology. Plugin developers have to use one of the stage types from the following four:

1. Stream Stage:

   This Stage/Component type is responsible for fetching the data from outer world into the processing system as a stream of data. This stage keeps feeding in the data in raw text form. Each line contains a row of data in such stream.

2. Process Stage:

   Process stage is used by plugin developers to develop components that perform distributed processing actions such as map-reduce actions, machine learning actions, perform filtering actions and similar others.

3. Database Stage:

   The database stage lets the end user save the incoming stream of data in raw or processed form depending on where the implemented component is placed in the topology. Each worker node in the distributed network can request a connection to the database server to save its raw/processed chunk of data.

4. Report Stage:

   Reporting stage helps the end user save data ready for analysis. The data that comes after flowing through all other stages is used by the report stage to generate a report scheme. A reporting scheme can be a live report of data for example, bar charts, block charts, donut charts, line charts etc. It can also be implemented in a way that shows the non-live persisted data in the form of static reports. The report stage doesn’t need the logic for out flow of data to be implemented as this stage is used only for visualizing data for analysis.

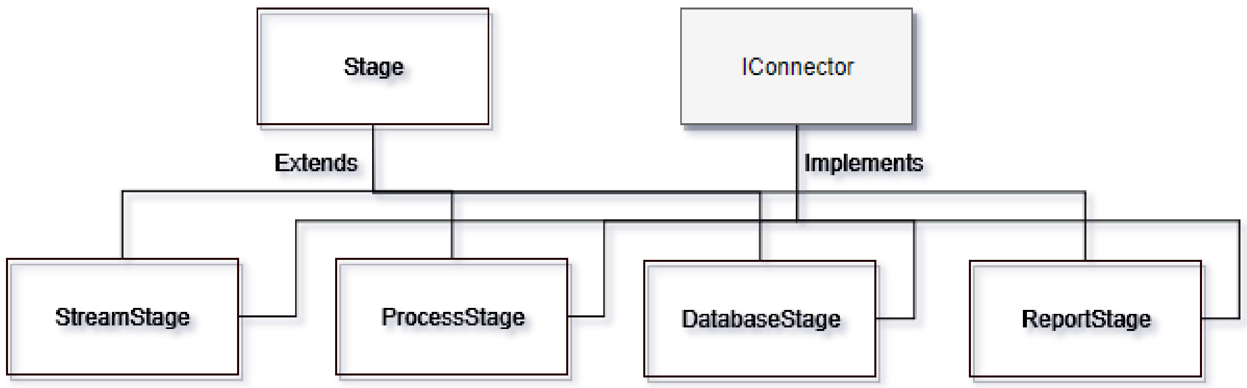


Fig. 6 Core Module Architecture.

Core module is what all the plugins are based on. All plugin developers need to link the core module library to their plugin projects so as to develop implementation of any stage.

3.1.3 Compiler Module

The compiler module is kind of an interpreter that interprets the incoming topology in the form of JSON and converts it into an actual working JAR job that can be executed by Apache Spark engine in its distributed network. The compiler is in itself an executable part of the system that is run on the backend to generate the corresponding JAR job for an input topology.

Compiler is generally invoked through the web GUI topology design area. When end user is done creating the topology, he/she can save the topology and press the “Compile” button to generate the output job in the backend. Compiler can also be used through the terminal and can directly be fed the topology to generate an Apache Spark job. It takes the path to the topology JSON as input and generates the output job to an output path which is also provided as an argument to it. When invoked through the web GUI, in the backend the system still calls the bash script to execute the compiler program over the specified topology.

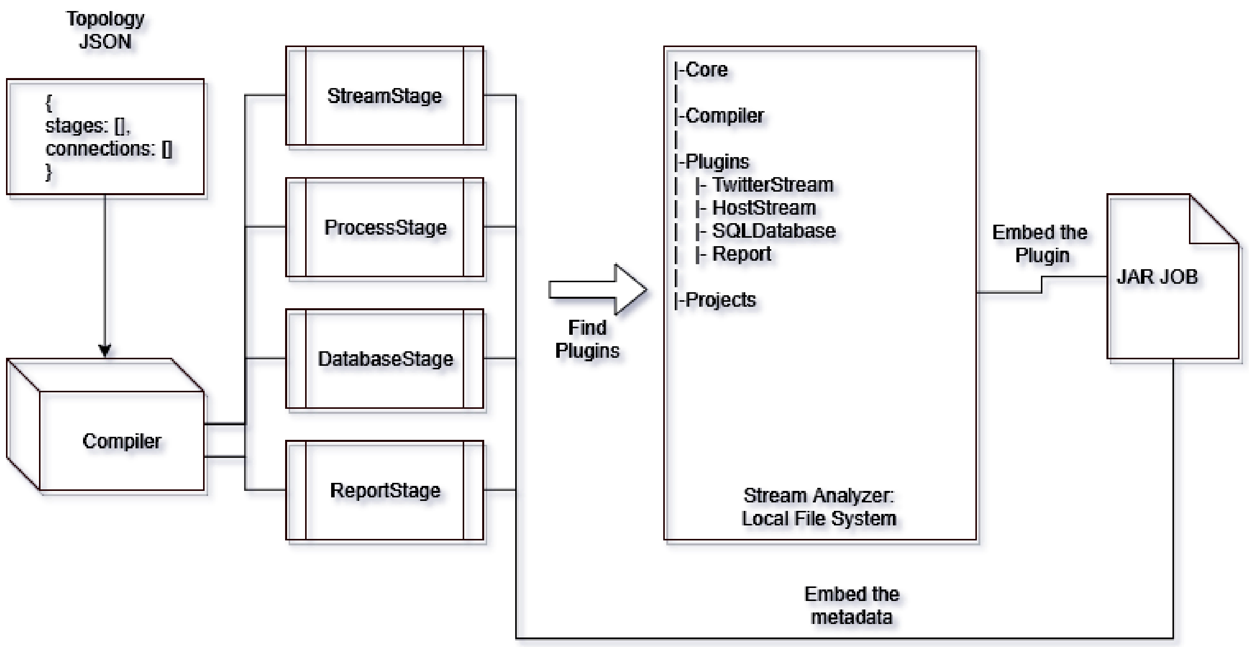


Fig. 7 Compiler Module

While interpreting the JSON topology, the compiler program traverses the whole topology and finds all possible components/stages present in the topology (as shown in Fig. 7). Each component/stage is a representation of a plugin on the web frontend and has its corresponding plugin implementation stored in the backend. Each such component found is used to search for and embed the plugin implementation code into the actual generated output job code. The metadata provided by the end user during topology creation to each component is also embedded into the job code as a serialized string. When the job is executed onto the Apache Spark engine, each node gets a copy of such metadata and de-serializes it so that the components can initialize themselves and perform in isolation on that node in the network.

3.1.4 Plugins Module

Plugins provide various means of extending a software system. A better system is considered one whose functionality can easily be changed or extended as per the needs. One of the main motivations behind developing Stream Analyzer was to have a distributed processing framework whose functionality can be extended by the end user himself.

The plugins module is a container for all the plugins that are represented by components on the frontend web GUI to the end users. Each plugin needs some metadata to preconfigure it before the actual execution of the Apache Spark job that is generated after compilation. During compilation, when the compiler encounters any component in the topology, it searches for the corresponding plugin in the local file system. Once found, the plugin is fed the metadata used to initialize itself in a serialized string form. This string is nothing but a serialized JSON object that is later de-serialized during job execution in the distributed network. Each plugin is a combination of three things:

1. Plug.json: The plugin configuration file. It tells about the plugin's main entry class file, how the plugin should be shown as a component on the web GUI and also tells. It also tells about if the component allows in and outflow of data from and to other components respectively.
2. Pom.xml: Maven configuration file for plugin project. As maven is used to define the library dependencies and to compile the code into actual JARs in the backend, this file is used to tell maven about how to compile the plugin and fetch the core module library as a dependency.
3. Program source code: Actual implementation of the plugin, which extends from the core module library.

3.2 Distributed Processing Frameworks & Local File System

Our software system heavily relies on distributed processing and storage frameworks, mainly Apache Spark and Apache Hadoop – HDFS. Apache Spark helps in execution of jobs among the nodes in a distributed network and Apache's Hadoop ecosystem's HDFS helps store the data over the distributed network.

3.2.1 Apache Spark

Incoming streams of data into the Stream Analyzer has to go through various stages to eventually be analyzed through a live report. The stages mentioned here are representations of small collections of actions to be performed over the streams of data for processing in a distributed manner.

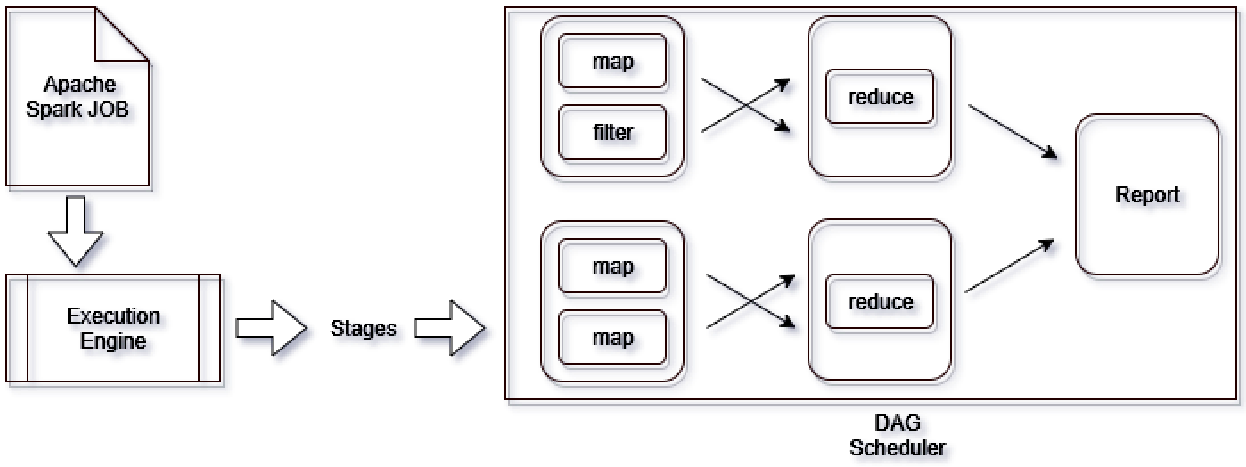


Fig. 8 Apache Spark's DAG Scheduling

The connection of stages itself depicts an abstract view of Apache Spark's directed acyclic graphs (DAGs). DAGs or directed acyclic graphs are generated by Apache's Spark engine, before actually executing a job in the distributed environment. The job is broken into small parts in such a way that there is no feedback of data (acyclic) back into any of the stages of the executing job. Each stream of data enters the system, gets processed and is finally stored into the storage nodes of the distributed processing environment (see Fig 8).

Stream Analyzer communicates with Apache Spark through the help of shell scripts, that are executed by the backend server of the web based GUI module. Just like the topology is compiled by the compiler when end user clicks on "*Compile*" button in the topology design area, the end user can click on "*Run*" button to perform the job simulation. The generated JAR file is submitted to Apache Spark through command line terminal. Apache Spark has its own tool, spark-submit to be specific, that takes as input the JAR job file and submits it to the already running Apache Spark's master node instance. From here, Apache Spark takes care of the execution of submitted job. It creates a directed acyclic graph, determines which nodes are present in the network for accepting the tasks for execution and sends the action/task to be performed in a serialized manner to those nodes for execution.

3.2.2 Apache Hadoop HDFS

The Stream Analyzer is dependent on Apache Hadoop's HDFS or Hadoop Distributed File System for storing the data generated by each stage and the metadata corresponding to each stage. The incoming streams of data are converted to RDDs (resilient distributed datasets) by the execution engine (i.e. Apache Spark) and are kept in memory for faster processing. But when there is a need for storing the data on small checkpoints of time, Apache Spark relies on Apache Hadoop's HDFS to store the RDDs on Hadoop storage nodes, as they are. Every next stage that needs the data as input from previous connected stage, looks for that data in the location on Hadoop file system (HDFS) where the previous stage saved the data at.

To communicate with the HDFS, Stream Analyzer uses the HDFS library provided by Apache Spark execution engine. The library provides API methods to talk to the file system and do basic operations such as list, store, read, remove and similar other.

To actually look into what each checkpoint-save for every stage has in HDFS store, we can use command line tool provided by Apache Hadoop, named "*hdfs.sh*". We execute the command as:

./hdfs.sh dfs -ls "/user/hadoop/projects/project-b02d234d-1f9c-4de0-a699-e2c77aa8f495/"

- hdfs.sh: the tool to do basic I/O operations on HDFS data nodes.
- dfs: sub-command of HDFS tool that tells Hadoop to specifically do disk related operations.
- -ls: this command is similar to LIST (ls) command of linux terminal. It lists the contents of a directory
- Path: the path to the location, which we want to list the contents of.

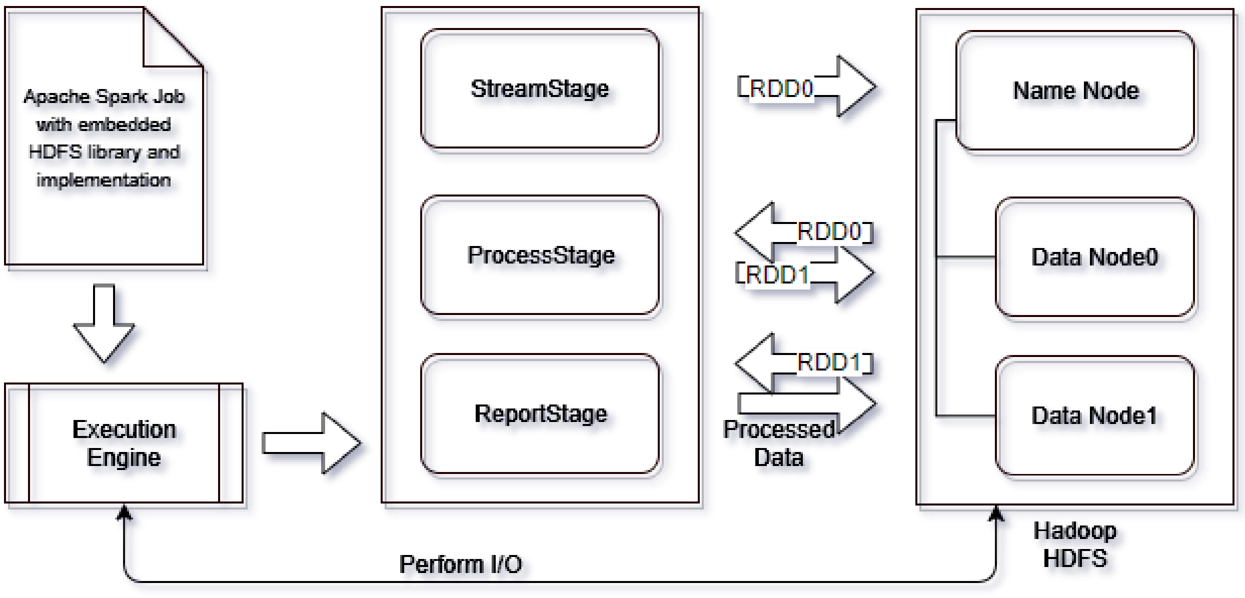


Fig. 9 Data Storage Process with Apache Spark and HDFS

As shown in the figure 9, each stage can store RDDs into data nodes of Hadoop eco system, take input data in the form of RDDs from the data nodes and can finally store the processed and analytical data back into Hadoop storage. Every stage has its associated folder in the Hadoop storage system. Every checkpoint data folder can be found at the location (URI):

"user/hadoop/projects/*{project-uuid}*/*{plugin_name}_{plugin_id}/*"

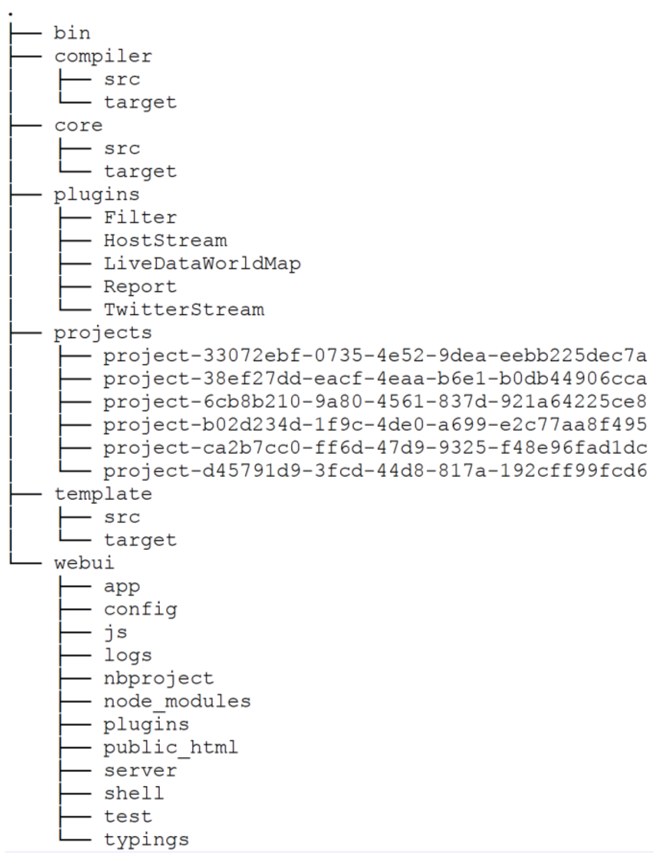
- project-uuid is the unique name of the project appended with a Universally Unique Identification Number.
- plugin_name is the actual name given to the implementation of a stage in the form of plugin and
- plugin_id is the id of the plugin that is generated in the topology design area. Every time a new compoenent is added to the topology, its ID is assigned to isolate it from other occurances of the same plugin in the topology.

3.2.3 Local File System

The local file system architecture for the software system in concern (i.e. Stream Analyzer), is a parent software project that holds together children program code in the form of modules. The basic structure looks like as shown in Fig. 10.

- bin folder holds shell scripts that are used to initialize the whole software ecosystem.
- compiler folder holds the logical code for the implementation of topology compiler.
- core folder holds the logical code related to core module
- plugins folder is used to store the plugin implementations of each stage type.
- projects folder is used to keep the projects created by end users and hold the topology design in the form of JSON file along with the logical code related to the actual functionality of corresponding stage type.

- template folder stores files that act as a template for creating a new project in the backend file system, every time an end user wants to create a topology in the web GUI design area.
- webui folder is there to hold the actual implementation of the web GUI. It is a NodeJS based web application which is run on the server side and keeps listening to the end user requests coming from the web frontend.



```
.
├── bin
├── compiler
│   ├── src
│   └── target
├── core
│   ├── src
│   └── target
├── plugins
│   ├── Filter
│   ├── HostStream
│   ├── LiveDataWorldMap
│   ├── Report
│   └── TwitterStream
├── projects
│   ├── project-33072ebf-0735-4e52-9dea-eebb225dec7a
│   ├── project-38ef27dd-eacf-4eaa-b6e1-b0db44906cca
│   ├── project-6cb8b210-9a80-4561-837d-921a64225ce8
│   ├── project-b02d234d-1f9c-4de0-a699-e2c77aa8f495
│   ├── project-ca2b7cc0-ff6d-47d9-9325-f48e96fad1dc
│   └── project-d45791d9-3fcd-44d8-817a-192cff99fcd6
├── template
│   ├── src
│   └── target
└── webui
    ├── app
    ├── config
    ├── js
    ├── logs
    ├── nbproject
    ├── node_modules
    ├── plugins
    ├── public_html
    ├── server
    ├── shell
    ├── test
    └── typings
```

Fig. 10 Stream Analyzer Directory Structure

4. SYSTEM IMPLEMENTATION

4.1 Materials used

All the materials needed are available freely online or from the University of Puerto Rico, Mayaguez campus (except the physical computing/data-storage nodes). Following are the tools and materials that we used to develop the system:

1. MEAN Stack

   MEAN is an abbreviation given to the combination of tools used to develop web based applications. MEAN stands for:

- M is for Mongo DB, which is a Database management software.
- E is for ExpressJS, which is a NodeJS based library and is used to create MVC architecture based web servers.
- A is for AngularJS, which is a JavaScript based library and is used to create beautifully fast web applications.
- N is for NodeJS, and is the base execution engine for JavaScript backend code.

Web developers can use MEAN software stack to create fast and responsive websites and web applications. It provides an MVC or Model-View-Controller programming paradigm with which, developer can modularize his/her web application code. Model depicts the schema representation of the entities in the program, View represents the actual viewable content by application users on the frontend and Controller represents the module that holds the actual backend logical code to server the user requests that come from the frontend web application.

Mongo DB is a well know No-SQL based database management software and is freely available online. We used the software to store end user's profile in object oriented manner with the help of entity schemas that we defined in our web application's logical code. We used ExpressJS to create the API endpoint hooks on the backend server. These endpoints are always listening to end user request, till the time the server is up and running.

AngularJS, has been extensively used throughout the frontend logical code in the web application. As mentioned, it is a JavaScript based library, it still manages to provide means of coding in object oriented manner, even though JavaScript is not an object oriented language. With the help of Typescript, which is new interpreted scripting language, developers can do object oriented programming while using JavaScript syntaxes. The Typescript coded application is actually first converted to actual JavaScript files and then served to the end user on the frontend. It helped us design and develop the GUI of our web application in a modularized manner.

NodeJS, is a JavaScript execution engine, but on the server side. We used ExpressJS to code our application's server logic. ExpressJS directly executes on the server side using NodeJS as an execution engine.

2. Apache Spark and Hadoop

   Apache Spark and Hadoop ecosystem has a bunch of libraries that help the distributed application developers make full use of these engines and develop applications that are 10x faster compared to the regular Hadoop based map-reduce jobs. These jobs can run on any platform where the execution engines (Spark and Hadoop) are capable of running. Basically the main requirement for these jobs to execute is need of Java Virtual Machines. Apache Spark and Hadoop are built with Java SDK and can run on almost all the devices which support Java. The core module is based on three main libraries which are:

- Spark Core lib

This library is the main JAR library provided by Apache Spark. It contains Classes and Interfaces to initialize an Apache Spark job. The library makes the spark job context be accessible to the job under execution with the help of "*JavaSparkContext*" class.

- Spark Streaming lib

  It aids the plugin developers in developing "*StreamStage*" implementations. Streaming plugins made with this library are fault tolerant and highly scalable. Furthermore, data streams created with this library, can be operated over by executing ad-hoc SQL queries.

- Spark SQL and Data frames library

  With this library, plugin developers are allowed to mix SQL queries into Spark jobs. It supports JDBC and ODBC connections and provides a common way to access any type of data source such as Apache Hive, Avro, Parquet, ORC and JSON. Apache Hadoop also provides many libraries to developers for creating Hadoop map-reduce jobs. But Apache Spark already has most of these libraries included itself and hence Spark job developers don't need to include them separately in their maven based project XML files. There are many more other libraries provided by Hadoop and our project uses one such library in the form on NodeJS based implementation. Our web GUI uses "*node-webhdfs*" library which is an abstraction/wrapper over Hadoop's HDFS library. We used this library in GUI to generate the actual live report by fetching the data from report stage's checkpoint directory and showing it on a live report web page inside our web application.

3. Java SDK 1.8

Java SDK is a freely available Software Development Kit (SDK) on the Internet and is used to develop applications using Java programming language. We are using JDK version 1.8 which is the latest one. Java 8 supports Lambda functions which help reduce the overall lines of code written for a Java application. Our whole software system (Stream Analyzer) is based on Java except the web frontend module, which is *"webui"*. We used NodeJS [JavaScript] for developing the frontend. But all other modules are programmed in Java programming language using the Java SDK 1.8. Even our execution engines, which are Apache Spark and Apache Hadoop, are solely based on Java.

4. Maven for Java Based Project Management

Apache Maven is a software project management tool based around the concept POM or Project Object Model. Maven provides a command line tool ("*mvn*") to perform project management operations over Java based projects and is in itself a Java based tool. It requires Java runtime to be installed on the machine. POM is basically a paradigm for managing and maintaining Java based project and their metadata with the help of well-structured dependencies description XML files. When a project is to be compiled and built, Maven searches for the "*pom.xml*" file in the project directory. This file gives information about the project (the metadata) and also tells about what are all the JAR library dependencies that this project is using. We used Maven to compile the topology created by end user in the topology design area, into a deployable Spark job with a "*.jar*" extension. When end user clicks on "*Compile*" button on the topology design area, the topology JSON is sent to the compiler module, which in turn executes a shell command for Maven based compilation. While compiling, maven traverses the "*pom.xml*" file present in topology's project directory and searches for all the dependencies present there. Each dependency's library JAR file is then embedded into the final output jar that corresponds to the topology itself. The output JAR is a deployable Spark job and contains all the required modules and libraries such as core module, Spark and Hadoop libraries and plugin module libraries (JARs) (See Fig. 11).

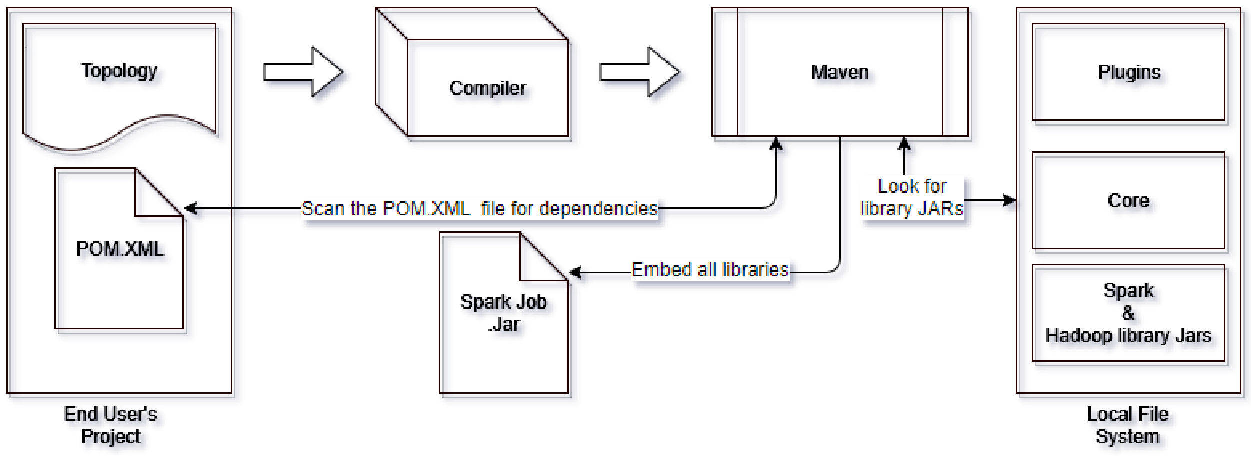


Fig. 11 Maven Based Compilation

4.2 Implementation

The Stream Analyzer has been implemented using Java, a widely used programming language and Typescript (JavaScript) which is scripting language and is mostly used for frontend web and mobile development. The Core, Compiler and Plugin modules have been developed using Java but the GUI (or the module webui) has been implemented using Typescript. The basic implementation architecture is shown in the figure 12 below.

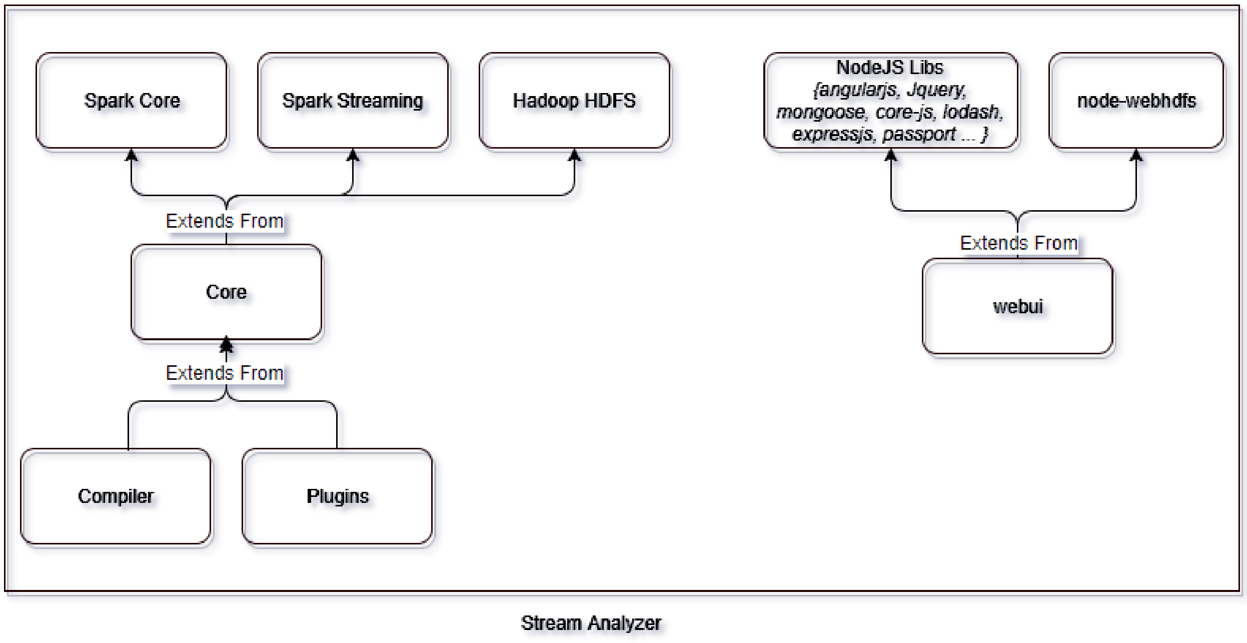


Fig. 12 Stream Analyzer Implementation Overview

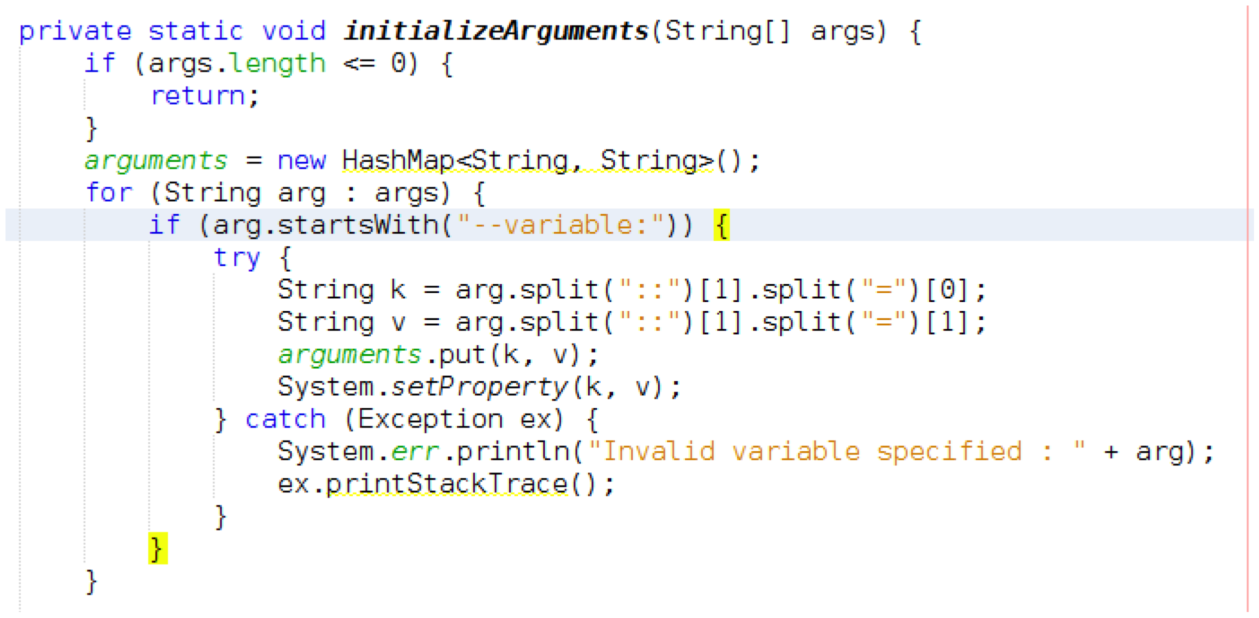
4.2.1 Implementation of Core module

The core module is what compiler and plugins modules are based on. The core module basically has two parts:

1. StreamAnalyzer class

   This class is used for the initialization of a spark job. The initializeArguments method of this class (see Fig. 13) takes as arguments the main arguments supplied to the job's entry point class (*Main class*) at the time of submitting the job to Apache Spark execution engine. The arguments are provided in the format: --*argumentName::argumentValue*. At the moment, the arguments supported by a job can be one of the following:

- *applicationId*: It is a unique job ID to be assigned to a job when submitting to spark engine.
- *applicationName*: A job name can also be supplied as an argument to the spark engine while submitting a job. It makes it easy to find a job in the list of submitted/running jobs on Apache Spark engine's UI.
- *hdfsMaster*: The URL to the Hadoop HDFS master name node. This helps the job to save checkpoint data by making a connection to Hadoop HDFS.
- *sparkMasterURL*: The URL to the Spark's master node. It should be provided in the form "*spark://master:port*", where master is the IP address of the master node and port is the port at which the master node is listening.
- *applicationPath*: This URI is used by a running job to save the checkpoint data in a particular path present in a Hadoop HDFS data node.



```
private static void initializeArguments(String[] args) {
    if (args.length <= 0) {
        return;
    }
    arguments = new HashMap<String, String>();
    for (String arg : args) {
        if (arg.startsWith("--variable:")) {
            try {
                String k = arg.split("::")[1].split("=")[0];
                String v = arg.split("::")[1].split("=")[1];
                arguments.put(k, v);
                System.setProperty(k, v);
            } catch (Exception ex) {
                System.err.println("Invalid variable specified : " + arg);
                ex.printStackTrace();
            }
        }
    }
```

Fig. 13 Job Initialization Subroutine

2. Stage Classes:

The stage classes are abstract classes which extend from "*Stage*" base class and implement the interface "*IConnector*". There are basic four Stage classes which are

- StreamStage: Used to implement data stream I/O plugins.
- ProcessStage: Used to implement actions such as Filter, Map, FlatMap, Aggregate etc. into plugins.
- DatabaseStage: Used to implement database storage plugins
- ReportStage: Used to Implement Live or static report plugins.

Every implementation of various stage type classes has to pass the metadata injected into its constructor to the super classes, which is "*Stage*" class. In Fig. 14, the constructor of "*Stage*" class takes the metadata as third argument. The first two arguments which are of types "*JavaSparkContext*" and "*StreamAnalyzer*" are injected through the entry point class of the spark job, when every such stage type implementation class is instantiated.

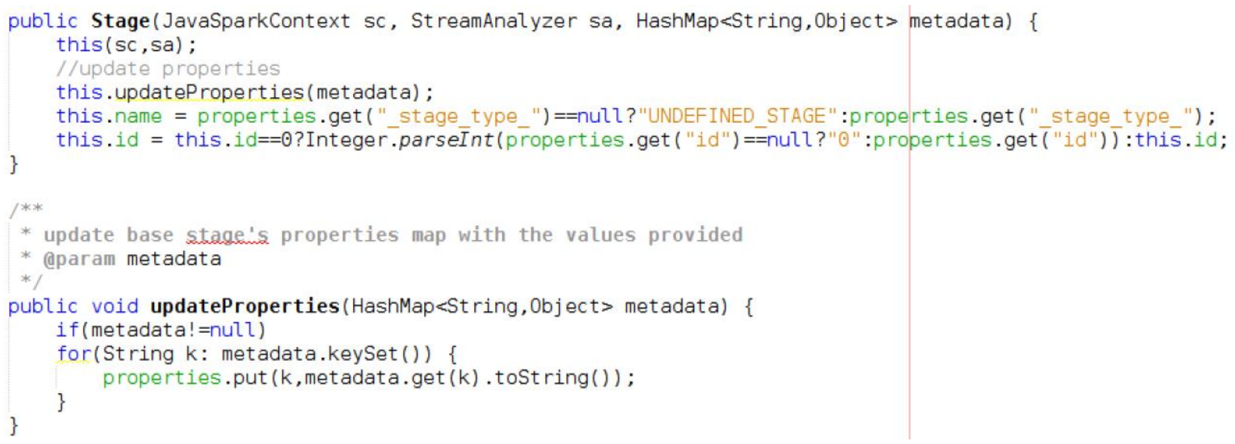

```java
public Stage(JavaSparkContext sc, StreamAnalyzer sa, HashMap<String,Object> metadata) {
    this(sc,sa);
    //update properties
    this.updateProperties(metadata);
    this.name = properties.get("_stage_type_")==null?"UNDEFINED_STAGE":properties.get("_stage_type_");
    this.id = this.id==0?Integer.parseInt(properties.get("id")==null?"0":properties.get("id")):this.id;
}

/**
 * update base stage's properties map with the values provided
 * @param metadata
 */
public void updateProperties(HashMap<String,Object> metadata) {
    if(metadata!=null)
    for(String k: metadata.keySet()) {
        properties.put(k,metadata.get(k).toString());
    }
}
```

Fig. 14 Stage Initialization

The "*updateProperties*" method (see Fig. 14) is used to update the "*properties*" hash map member variable with the metadata key value pairs that the end user specifies during configuration of the stage on the topology design area.

4.2.2 Implementation of Compiler module

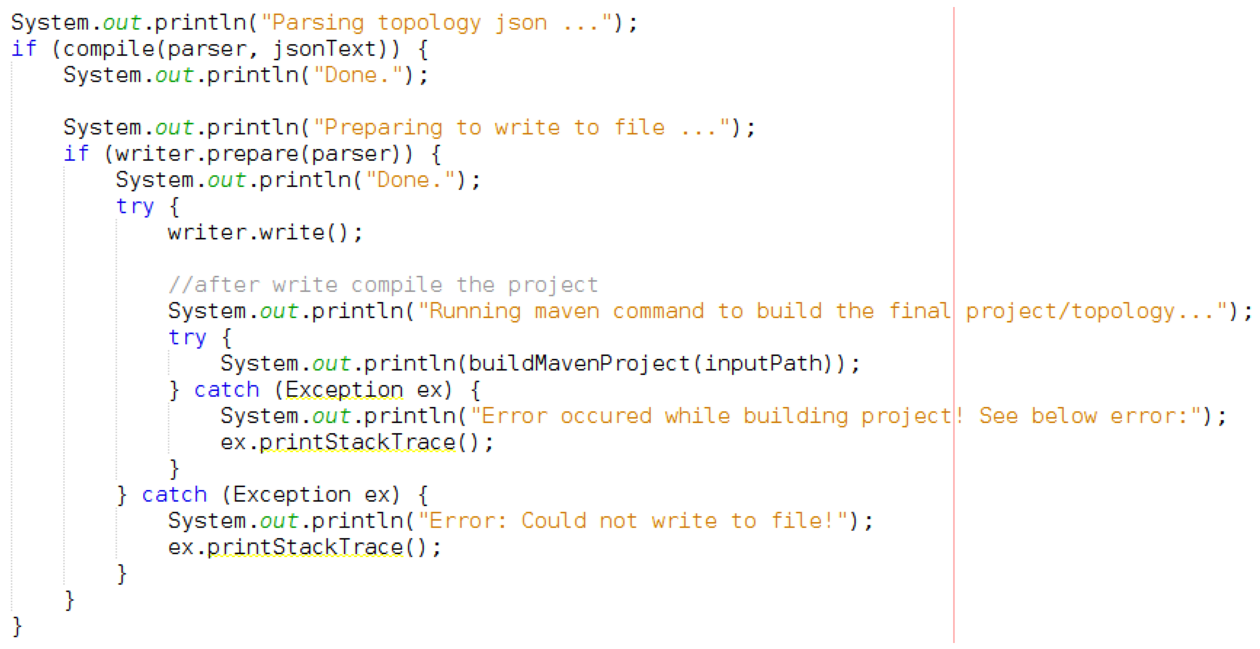
The compiler module plays a major role in conversion of a job topology to actually deployable spark job JAR file. It is made up of following sub-modules:

1. Command Line Interface (CLI):

   The CLI is the main entry point for starting the execution of compiler. It invokes the subroutines required to generate a fully deployable spark job. It has been implemented into following classes:

   - *Main*: The Main class contains the actual entry point method which accepts the command line arguments and passes them down to the Compiler class object. The arguments supported by the CLI are "*verbose*" and "*compile*". "*Verbose*" command line argument tells the compiler to output the compilation process steps on terminal screen. This output can also be seen on the web GUI while compiling a topology into a spark job. The "*compile*" command line argument tells the CLI to instantiate the "*Compiler*" class object and start compiling the input topology. There are two more important arguments to be supplied to the CLI which are "*inputPath*" and "*outputPath*". The "*inputPath*" argument tells the compiler the exact path of the project/topology to be compiled and the "*outputPath*" argument tells the compiler the exact path where the output JAR is to be saved.

- *Compiler*: The "*Compiler*" class has two important methods that perform the interpretation and then actual compilation using Apache Maven tool to generate final spark job JAR. The "*compile*" method is responsible for parsing the various stages present in a job topology. This method reads the "*topology.json*" file present in the "*inputPath*" location. It then uses the "*Parser*" class object to parse the stages from JSON file into "*jsonText*" member variable and convert them into instantiated stage entity classes. It also instantiates a "*ClassWriter*" class object to write logic, corresponding to the topology under compilation, to the entry point main class of the generated spark job JAR.



```
System.out.println("Parsing topology json ...");
if (compile(parser, jsonText)) {
    System.out.println("Done.");

    System.out.println("Preparing to write to file ...");
    if (writer.prepare(parser)) {
        System.out.println("Done.");
        try {
            writer.write();

            //after write compile the project
            System.out.println("Running maven command to build the final project/topology...");
            try {
                System.out.println(buildMavenProject(inputPath));
            } catch (Exception ex) {
                System.out.println("Error occured while building project! See below error:");
                ex.printStackTrace();
            }
        } catch (Exception ex) {
            System.out.println("Error: Could not write to file!");
            ex.printStackTrace();
        }
    }
}
```

Fig. 15 Compilation Method

2. Entities:

The Entity classes are nothing but plain old java objects (POJOs) that hold the property values for each stage type implementation. When the stages parser is done with parsing the stages read from a "*topology.json*" file, they are directly mapped to these entities. The values are copied to the each corresponding field in a stage type entity object from the topology. Entities have been implemented as:

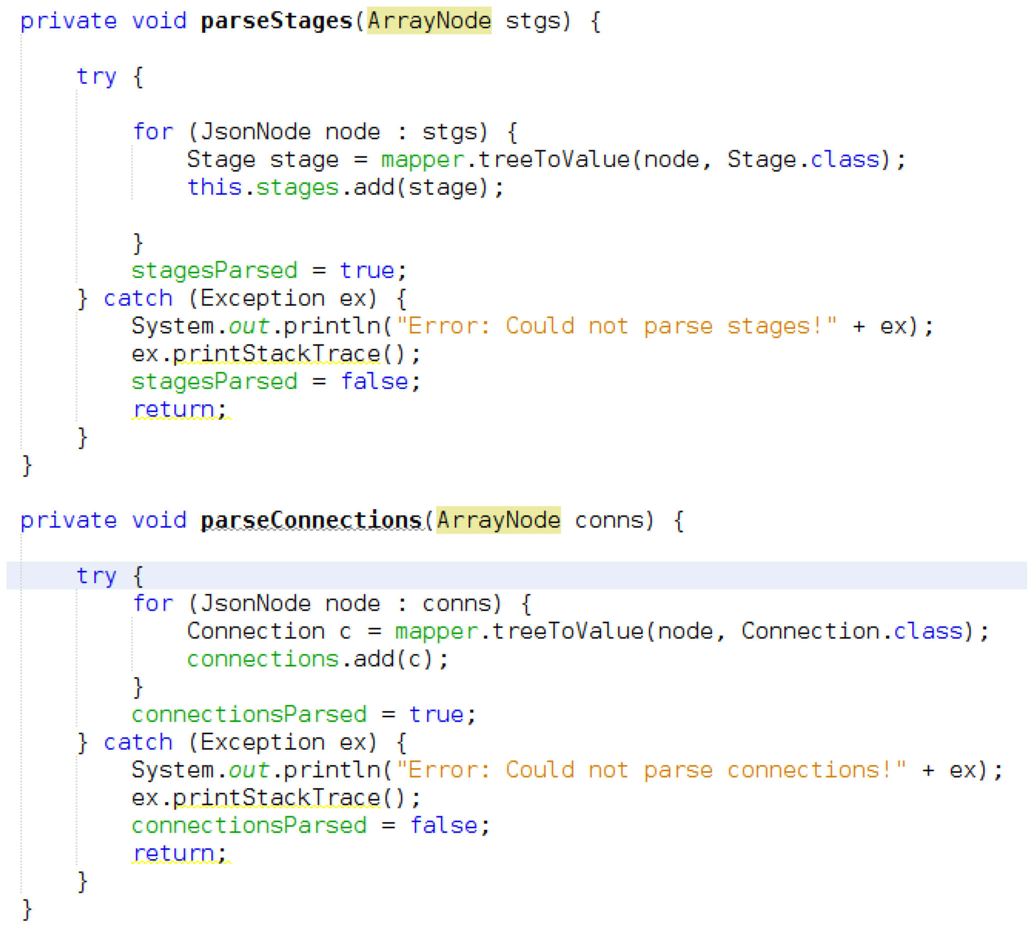
- Stage Entity: This class is a POJO class. Metadata fields and stage type fields are mapped on to this class's instantiated object.
- Connection Entity: This POJO class's instance is mapped with the connection related properties of each stage in the topology. If there is a connection or in/out flow of data between stages, a connection object is created and metadata about the connection is injected into it.

- Plugin Entity: This POJO class holds properties related to each stage's plugin implementation.

3. Parser:

   When the Compiler is done reading the input topology file from the "*inputPath*", it passes the read string of data to the "*parse*" method of Parser class instance. The "*parse*" method reads the whole JSON tree in the form of string and separates the stages and connections among them into different "*ArrayNode*" instances. An "*ArrayNode*" class is used to represent JSON string trees in object oriented manner. These different stages and connections array nodes are passed to "*parseStages*" and "*parseConnections*" methods respectively (See Fig. 16).

- parseStages method reads the stages "*ArrayNode*" JSON tree and using the "*ObjectMapper*" class instance, maps the stages directly to the various stage type entity class instances. These stage instances are added to the final stages array.
- parseConnections method reads the connections "*ArrayNode*" JSON tree and uses the same "*ObjectMapper*" class instance as "*parseStages*" method to map the connections directly to the connection entity class instances. These connection instances are added to the final connections array.



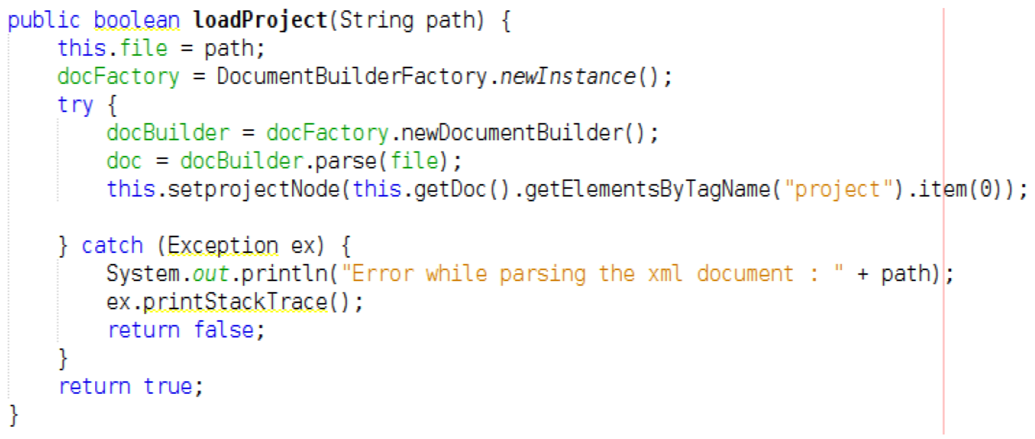
```
private void parseStages(ArrayNode stgs) {

    try {

        for (JsonNode node : stgs) {
            Stage stage = mapper.treeToValue(node, Stage.class);
            this.stages.add(stage);

        }
        stagesParsed = true;
    } catch (Exception ex) {
        System.out.println("Error: Could not parse stages!" + ex);
        ex.printStackTrace();
        stagesParsed = false;
        return;
    }
}

private void parseConnections(ArrayNode conns) {

    try {
        for (JsonNode node : conns) {
            Connection c = mapper.treeToValue(node, Connection.class);
            connections.add(c);
        }
        connectionsParsed = true;
    } catch (Exception ex) {
        System.out.println("Error: Could not parse connections!" + ex);
        ex.printStackTrace();
        connectionsParsed = false;
        return;
    }
}
```

Fig. 16 Parsing Methods

4. POMEditor:

“*POMEditor*” class is responsible for managing the end user’s project’s management. It is used by the “*ClassWriter*” class instance to add, remove or update dependencies of any java project which is internally managed by Apache Maven and has a “*pom.xml*” file in it. It provides following API methods for managing POM xml files:

- loadProject method helps load the pom.xml file from the project’s root directory. A “*pom.xml*” file contains XML document that describes the project along with its dependencies. Each project has some unique attributes attached to it which are, "*artifactId*", "*groupId*" and "*version*". These attributes help maven to keep registry of such projects. Each dependency also has these attributes which help maven identify that which specific library has to be embedded in to the eventually generated spark job JAR file. As shown in the Fig. 17, the "*loadProject*" methods takes in the path to POM XML file as argument and loads the XML document architecture with the help of "*DocumentBuilderFactory*" instance.



```
public boolean loadProject(String path) {
    this.file = path;
    docFactory = DocumentBuilderFactory.newInstance();
    try {
        docBuilder = docFactory.newDocumentBuilder();
        doc = docBuilder.parse(file);
        this.setprojectNode(this.getDoc().getElementsByTagName("project").item(0));

    } catch (Exception ex) {
        System.out.println("Error while parsing the xml document : " + path);
        ex.printStackTrace();
        return false;
    }
    return true;
}
```

Fig. 17 POM XML Loading Method

- addProjectDependency/removeProjectDependency method is used to modify the POM XML file and insert project dependency information in it. It searches for the "<*dependencies*>" XML tag in the file and places/removes a dependency corresponding to each stage type plugin implementation. Each plugin is a maven project in itself, hence it must also have "*artifactId*", "*groupId*" and "*version*" attributes, so that while compiling, maven can search for these registered plugins in its repository and embed/detach them into/from the finally generated spark job JAR file.

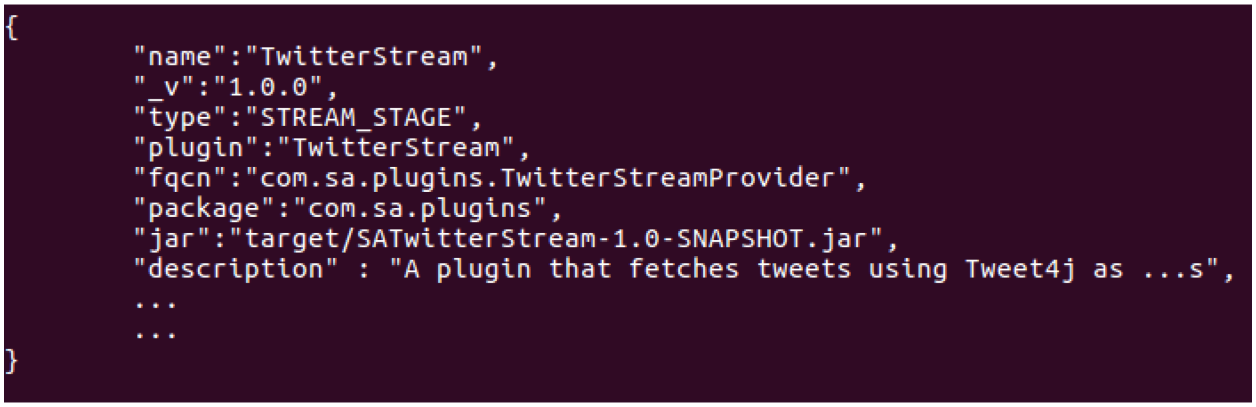
4.2.3 Implementation of Plugins module

Plugins are basically used to extend functionality of a software system. Stream Analyzer is majorly made up of plugins. Each stage type (stream stage, process stage etc.) has a visual representation in the form of component when viewed on the topology design area and the same stage is implemented behind the scenes in the form of plugins, which are maven based java projects and extend from the core module. For the sake of this demonstration version of Stream Analyzer, we developed three main plugins for performing map reduce actions on live streams of data. These plugins are

1. TwitterStreamProvider Plugin

The "*TwitterStreamProvider*" class is the main entry point for this plugin. It extends from the core module's "*StreamStage*" class. The project POM XML file corresponding to this plugin has "*Twitter4J*" library as dependency too because this library provides API methods to fetch live tweets from throughout the world as an input stream of data.

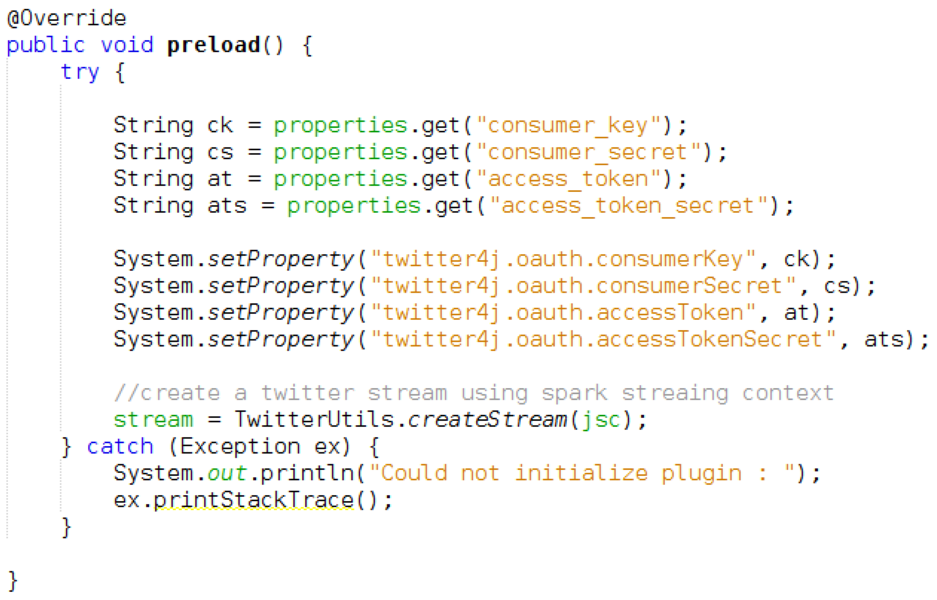
Each plugin project also has a "*plug.json*" file which describes the frontend behavior of the plugin's component representation (in the topology design area). As shown in the Fig. 18, the "*plug.json*" file includes plugin's name, version, stage type, fully qualified class name for entry point, package name, the implementation JAR library path, description of the plugin etc.



```
{
        "name":"TwitterStream",
        "_v":"1.0.0",
        "type":"STREAM_STAGE",
        "plugin":"TwitterStream",
        "fqcn":"com.sa.plugins.TwitterStreamProvider",
        "package":"com.sa.plugins",
        "jar":"target/SATwitterStream-1.0-SNAPSHOT.jar",
        "description" : "A plugin that fetches tweets using Tweet4j as ...s",
        ...
        ...
}
```

Fig. 18 Plugin Description File

When the topology is parsed and metadata is extracted from it by the compiler, it is serialized and fed into the plugin implementation at runtime. The plugin has to de-serialize the metadata using the "*deserialize*" method present in the base class "*Stage*" of the core module. The injected metadata is mapped to "*properties*" hash map member variable of the base class "*Stage*", and is eventually used after deserialization in the "*preload*" method of the plugin for initialization purpose (see Fig. 19).



```
@Override
public void preload() {
    try {

        String ck = properties.get("consumer_key");
        String cs = properties.get("consumer_secret");
        String at = properties.get("access_token");
        String ats = properties.get("access_token_secret");

        System.setProperty("twitter4j.oauth.consumerKey", ck);
        System.setProperty("twitter4j.oauth.consumerSecret", cs);
        System.setProperty("twitter4j.oauth.accessToken", at);
        System.setProperty("twitter4j.oauth.accessTokenSecret", ats);

        //create a twitter stream using spark streaing context
        stream = TwitterUtils.createStream(jsc);
    } catch (Exception ex) {
        System.out.println("Could not initialize plugin : ");
        ex.printStackTrace();
    }

}
```

Fig. 19 Plugin Initialization method

The plugin implementation's entry point class also has to implement the abstract methods from the "*Stage*" class and "*IConnector*" interface. The "*start*" method of plugin's entry point class is responsible for starting the actual processing of incoming data stream at this stage in the job's topology (see Fig. 20).

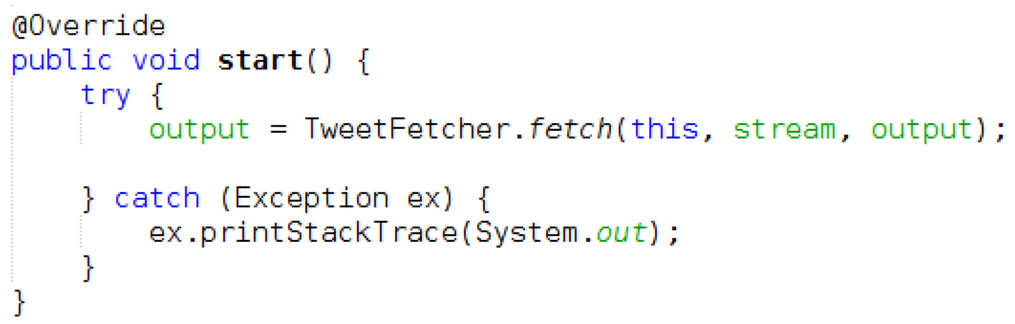
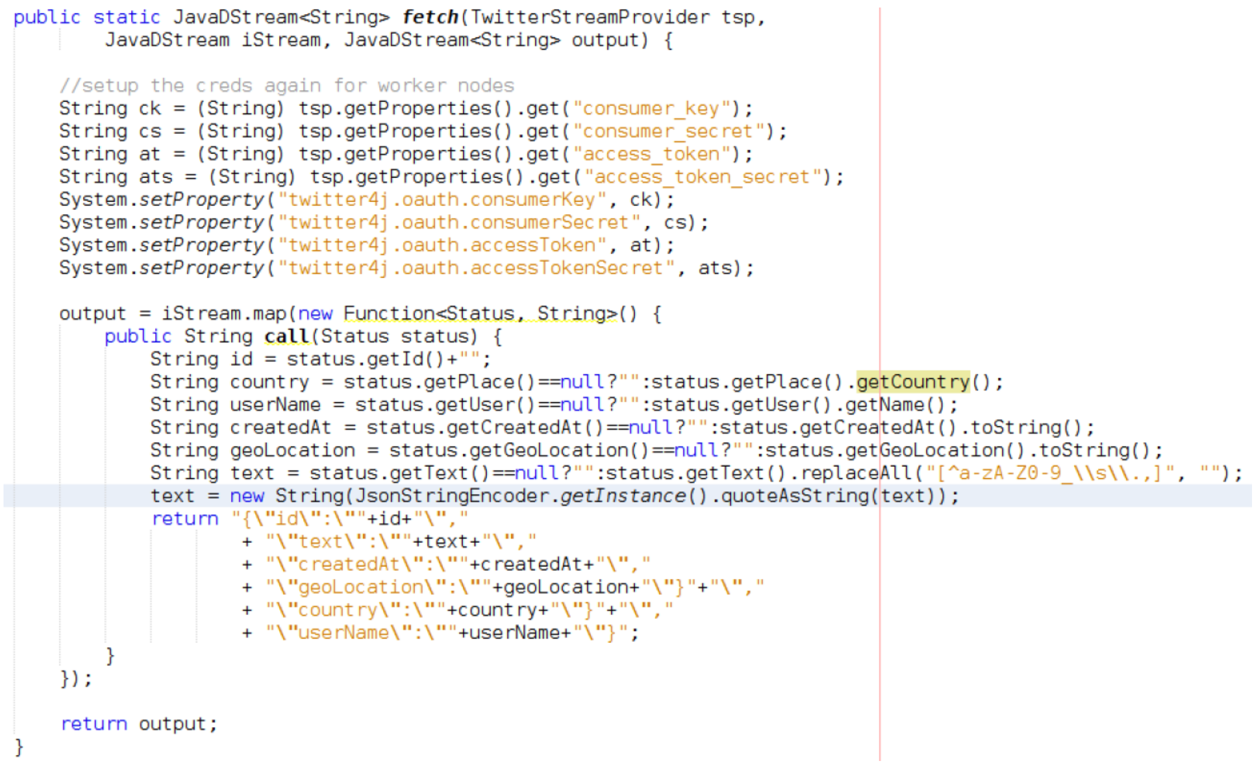

```
@Override
public void start() {
    try {
        output = TweetFetcher.fetch(this, stream, output);

    } catch (Exception ex) {
        ex.printStackTrace(System.out);
    }
}
```

Fig. 20 Method to call Action Class

Each plugin also has to implement another class, an action class, that implements "*Serializable*" interface and an object of this class should be instantiated in the entry point class of the plugin or the method that performs the actions. Action class is needed by apache spark engine to serialize and send the action to be performed to each node in its distributed processing network. Action classes basically should have just one main method that holds the logical code to perform action over the incoming stream of data. As shown in Fig. 21, the method "*fetch*" is a member of action class "*TweetsFetcher*", which is also a part of the "*TwitterStreamProvider*" plugin module. This method is static and is directly called by the "*start*" method of the "*TwitterStreamProvider*" class. The "*fetch*" method performs the basic "*map*" action over the incoming stream of data as

“*JavaDStream*”. It basically converts the raw tweets in the incoming data stream to JSON formatted string, which in turn can be fed into the next stages in the topology.



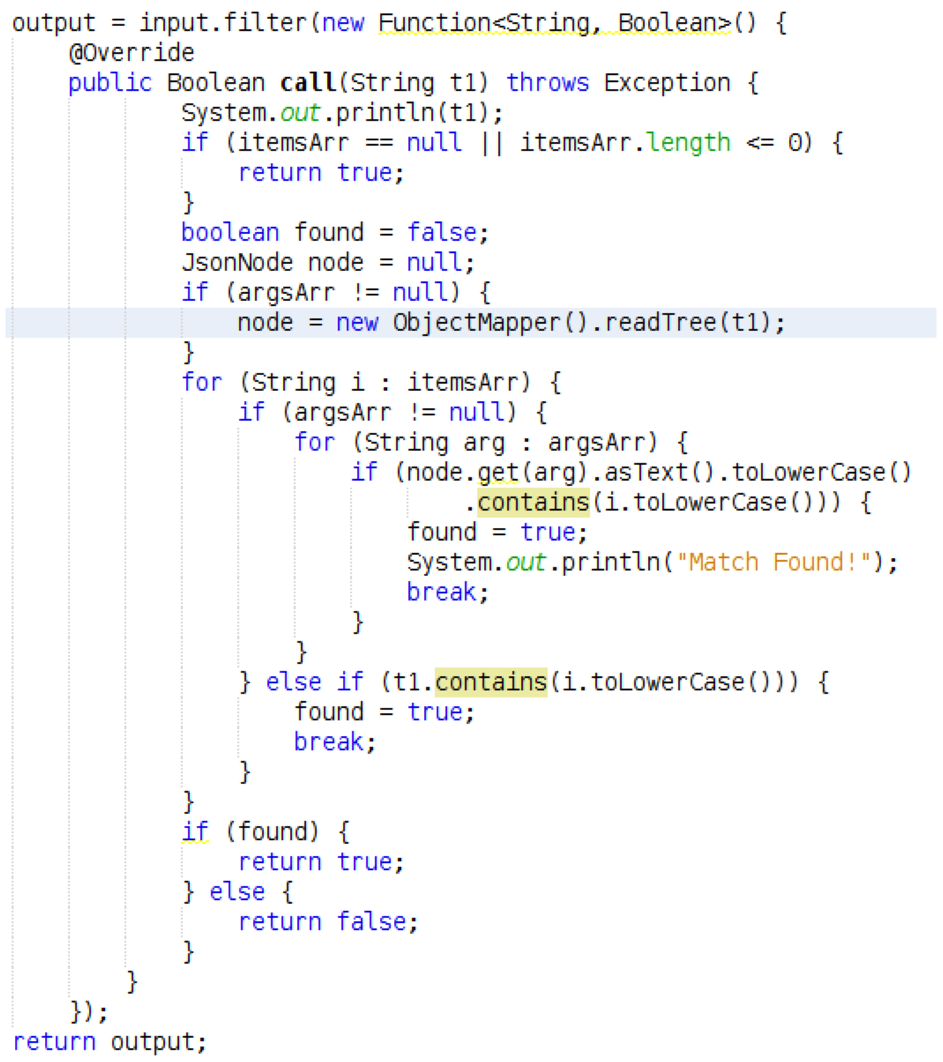
```java
public static JavaDStream<String> fetch(TwitterStreamProvider tsp,
        JavaDStream iStream, JavaDStream<String> output) {

    //setup the creds again for worker nodes
    String ck = (String) tsp.getProperties().get("consumer_key");
    String cs = (String) tsp.getProperties().get("consumer_secret");
    String at = (String) tsp.getProperties().get("access_token");
    String ats = (String) tsp.getProperties().get("access_token_secret");
    System.setProperty("twitter4j.oauth.consumerKey", ck);
    System.setProperty("twitter4j.oauth.consumerSecret", cs);
    System.setProperty("twitter4j.oauth.accessToken", at);
    System.setProperty("twitter4j.oauth.accessTokenSecret", ats);

    output = iStream.map(new Function<Status, String>() {
        public String call(Status status) {
            String id = status.getId()+"";
            String country = status.getPlace()==null?"":status.getPlace().getCountry();
            String userName = status.getUser()==null?"":status.getUser().getName();
            String createdAt = status.getCreatedAt()==null?"":status.getCreatedAt().toString();
            String geoLocation = status.getGeoLocation()==null?"":status.getGeoLocation().toString();
            String text = status.getText()==null?"":status.getText().replaceAll("[^a-zA-Z0-9_\\s\\.,]", "");
            text = new String(JsonStringEncoder.getInstance().quoteAsString(text));
            return "{\"id\":\""+id+"\","
                    + "\"text\":\""+text+"\","
                    + "\"createdAt\":\""+createdAt+"\","
                    + "\"geoLocation\":\""+geoLocation+"\"}"+"\","
                    + "\"country\":\""+country+"\"}"+"\","
                    + "\"userName\":\""+userName+"\"}";
        }
    });

    return output;
}
```

Fig. 21 Method to perform Action over Data Stream

2. Filter Plugin

The Filter plugin is an implementation of “*ProcessStage*” component. It is used to filter out only those rows of data from the incoming data stream, which have a particular field value, matching the one specified by the end user during topology design. It is very similar in implementation to that of “*TwitterStreamProvider*”. The only difference lies in its action class, in which, instead of using “*map*” action, we are doing a “*filter*” operation as shown in the Fig. 22.

The output of “*filter*” operation contains only those rows of data where the “*call*” method of the procedure, passed as an instance of “*Function*” class (provided by apache spark) to this operation (filter), returns true. All the unmatched rows will be stripped from the final output.

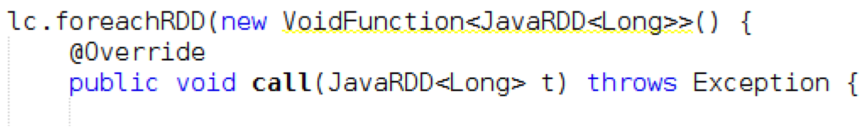


```
output = input.filter(new Function<String, Boolean>() {
    @Override
    public Boolean call(String t1) throws Exception {
            System.out.println(t1);
            if (itemsArr == null || itemsArr.length <= 0) {
                return true;
            }
            boolean found = false;
            JsonNode node = null;
            if (argsArr != null) {
                node = new ObjectMapper().readTree(t1);
            }
            for (String i : itemsArr) {
                if (argsArr != null) {
                    for (String arg : argsArr) {
                        if (node.get(arg).asText().toLowerCase()
                                .contains(i.toLowerCase())) {
                            found = true;
                            System.out.println("Match Found!");
                            break;
                        }
                    }
                } else if (t1.contains(i.toLowerCase())) {
                    found = true;
                    break;
                }
            }
            if (found) {
                return true;
            } else {
                return false;
            }
        }
    });
return output;
```

Fig. 22 Filter Operation

3. Report Plugin

The Report plugin is an implementation of the “*ReportStage*” component. It is used to perform the reduce operation over mapped data streams. It has a similar architecture as other plugins, the main difference is in the plugin’s action class, in which, the action method performs iteration over the incoming data stream by using the method "*foreachRDD*". This method is provided by apache spark core library and is a member of input data stream class "*JavaDStream*". As shown in Fig. 23, it iterates over each RDD present in the input stream and calculates the final output, which it then stores back to Hadoop's distribute file system (HDFS). It uses the "*FileSystem*" class to do basic file system I/O on Hadoop data node. This class is provided by Hadoop's core library.



```
lc.foreachRDD(new VoidFunction<JavaRDD<Long>>() {
    @Override
    public void call(JavaRDD<Long> t) throws Exception {
```

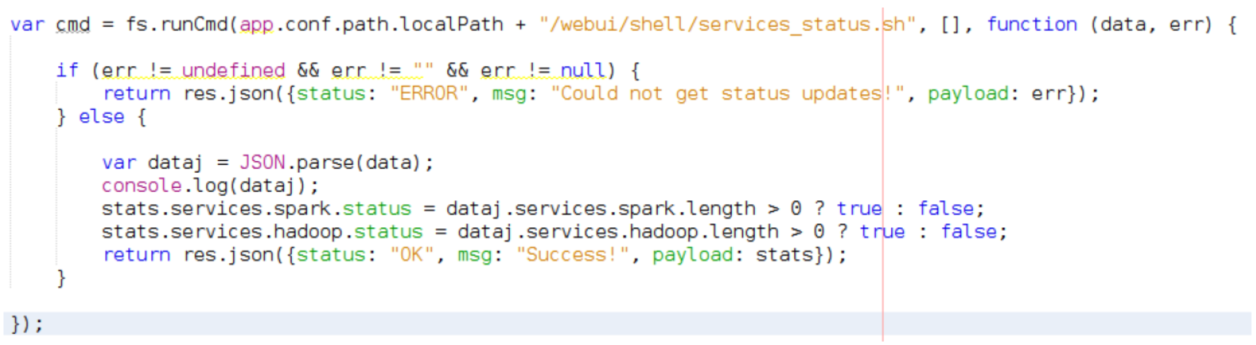
Fig. 23 Reduce operation for Report generation

4.2.4 Implementation of GUI (webui) module

"*webui*" module contains majority of the logical code that is used to present an interface to the end user where the user can manage the projects, design job topologies, compile the jobs, run them and view the live report generated when the job is executed. It consists of two major parts, client side and server side. The client side is what's the end user is able to access. Each request made by the end user while interacting with the client side is sent to the server side to be processed. The calls made are in the form of JSON and are made with the help of basic REST requests such as create, read, update and delete (CRUD). The major parts of web GUI module include:

1. Dashboard – Home sub-module

   The "*Dashboard*" sub-module contains the client-server side code for the viewable dashboard area of the GUI. We implemented the dashboard to show current projects associated to the user's account along with their statuses (running or stopped). The dashboard shows the statuses of the two major frameworks Apache Spark and Apache Hadoop. To get the statuses, the client side sends a request to the backend server, the server processes the request by sensing the statuses of the two framework along with sensing the current running/stopped status of the projects/jobs shown on the dashboard.

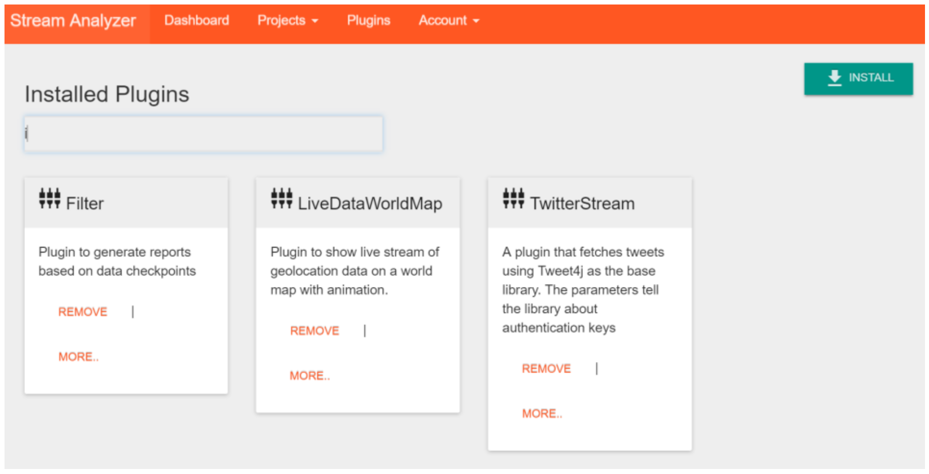


```
var cmd = fs.runCmd(app.conf.path.localPath + "/webui/shell/services_status.sh", [], function (data, err) {

    if (err != undefined && err != "" && err != null) {
        return res.json({status: "ERROR", msg: "Could not get status updates!", payload: err});
    } else {

        var dataj = JSON.parse(data);
        console.log(dataj);
        stats.services.spark.status = dataj.services.spark.length > 0 ? true : false;
        stats.services.hadoop.status = dataj.services.hadoop.length > 0 ? true : false;
        return res.json({status: "OK", msg: "Success!", payload: stats});
    }

});
```

Fig. 24 Part of subroutine to get the statuses of projects and frameworks

2. Plugins sub-module

The "*plugins*" sub-module also has the client and server side implementation in the "*webui*" module. The plugins viewable area on the client side GUI is used to get information about already installed plugins and also allows the end user to install new plugin implementations from their local hard-disk in the form of an archive as a "*.zip*" file format.



Installed Plugins

Filter

Plugin to generate reports based on data checkpoints

LiveDataWorldMap

Plugin to show live stream of geolocation data on a world map with animation.

TwitterStream

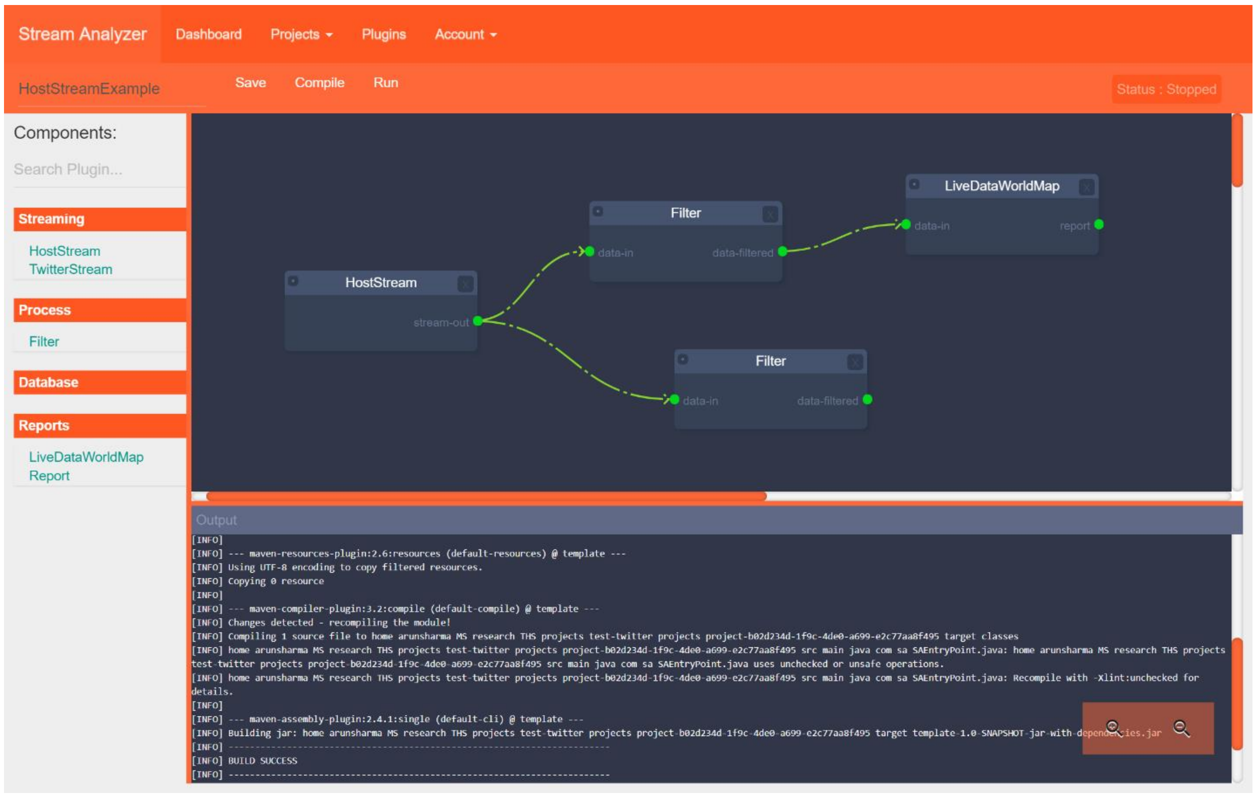
A plugin that fetches tweets using Tweet4j as the base library. The parameters tell the library about authentication keys

Fig. 25 Plugins web-page

Already installed plugins can be seen on the plugins viewable area and can also be searched for by typing the desired plugin's name in the text as show in the Fig. 25. Plugins can be installed by clicking the "*Install*" button on the top right corner of the plugins page, which asks for the location of the plugin zip file on user's hard disk. Once installed, plugin is shown on the plugins viewable area, and also on the topology design area. Plugins are loaded as a JSON object during the initialization of the "*webui*" module. When requested by the client side, the same JSON array is returned from the server side.

3. Projects sub-module

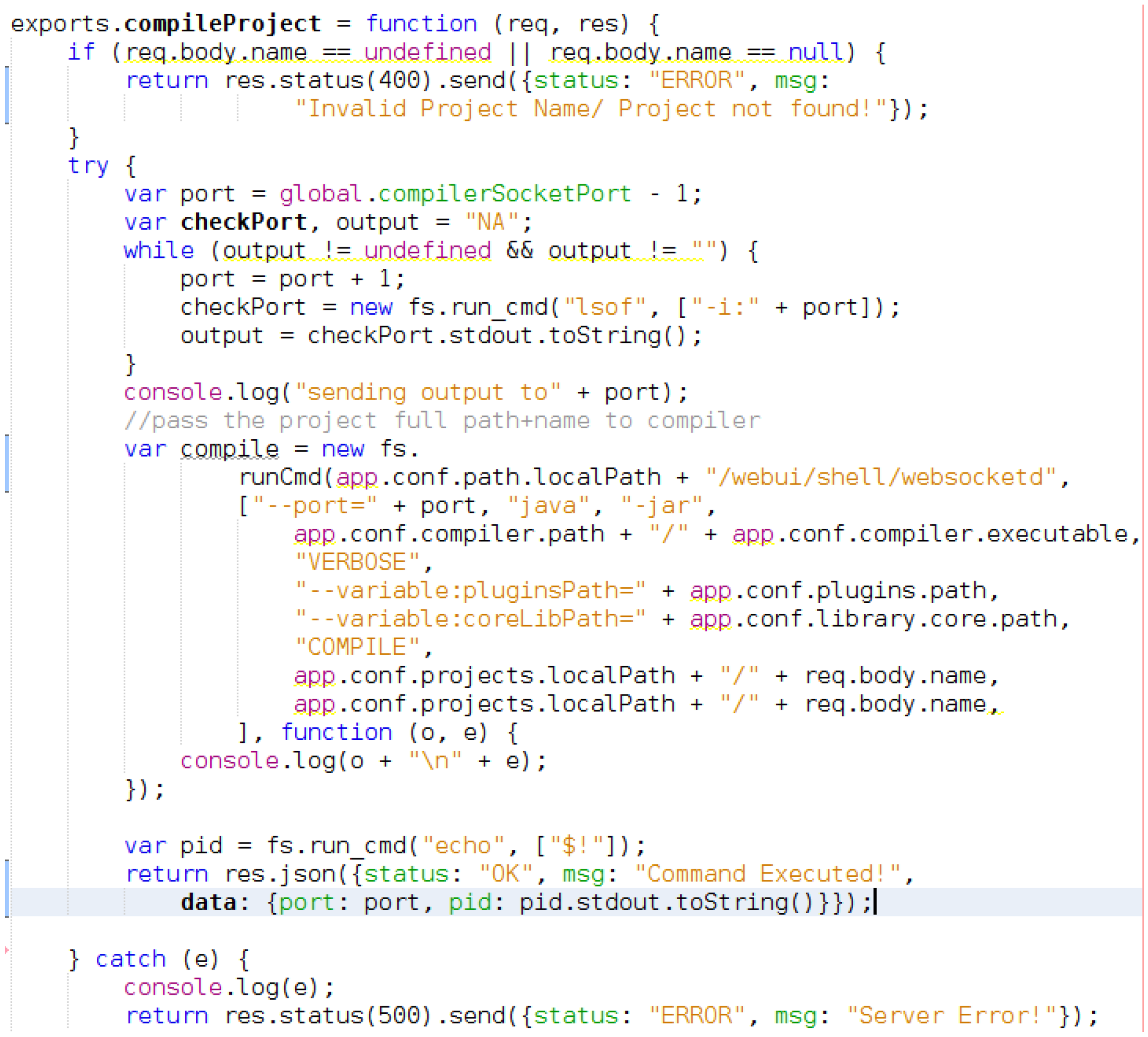
Projects sub-module contains the logical code that deals with managing the projects and job topologies. Job topologies are actually a part of a user created project. Each project is allowed to have only one topology. Topology is created on the topology design area (see Fig. 26).



```
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ template ---
[INFO] Using UTF-8 encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.2:compile (default-compile) @ template ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to home arunsharma MS research THS projects test-twitter projects project-b02d234d-1f9c-4de0-a699-e2c77aa8f495 target classes
[INFO] home arunsharma MS research THS projects test-twitter projects project-b02d234d-1f9c-4de0-a699-e2c77aa8f495 src main java com sa SAEntryPoint.java: home arunsharma MS research THS projects
test-twitter projects project-b02d234d-1f9c-4de0-a699-e2c77aa8f495 src main java com sa SAEntryPoint.java uses unchecked or unsafe operations.
[INFO] home arunsharma MS research THS projects test-twitter projects project-b02d234d-1f9c-4de0-a699-e2c77aa8f495 src main java com sa SAEntryPoint.java: Recompile with -Xlint:unchecked for
details.
[INFO]
[INFO] --- maven-assembly-plugin:2.4.1:single (default-cli) @ template ---
[INFO] Building jar: home arunsharma MS research THS projects test-twitter projects project-b02d234d-1f9c-4de0-a699-e2c77aa8f495 target template-1.0-SNAPSHOT-jar-with-dependencies.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Fig. 26 Topology design area

Each block represents a plugin implementation and is referred to as a component on the topology design area. The connection between the components shows the in/out flow of data from one stage/component to other. User can add multiple such components by clicking on a particular component listing in the left pane. The output generated during compilation or running of the job can be seen in the terminal output pane in bottom of the page.



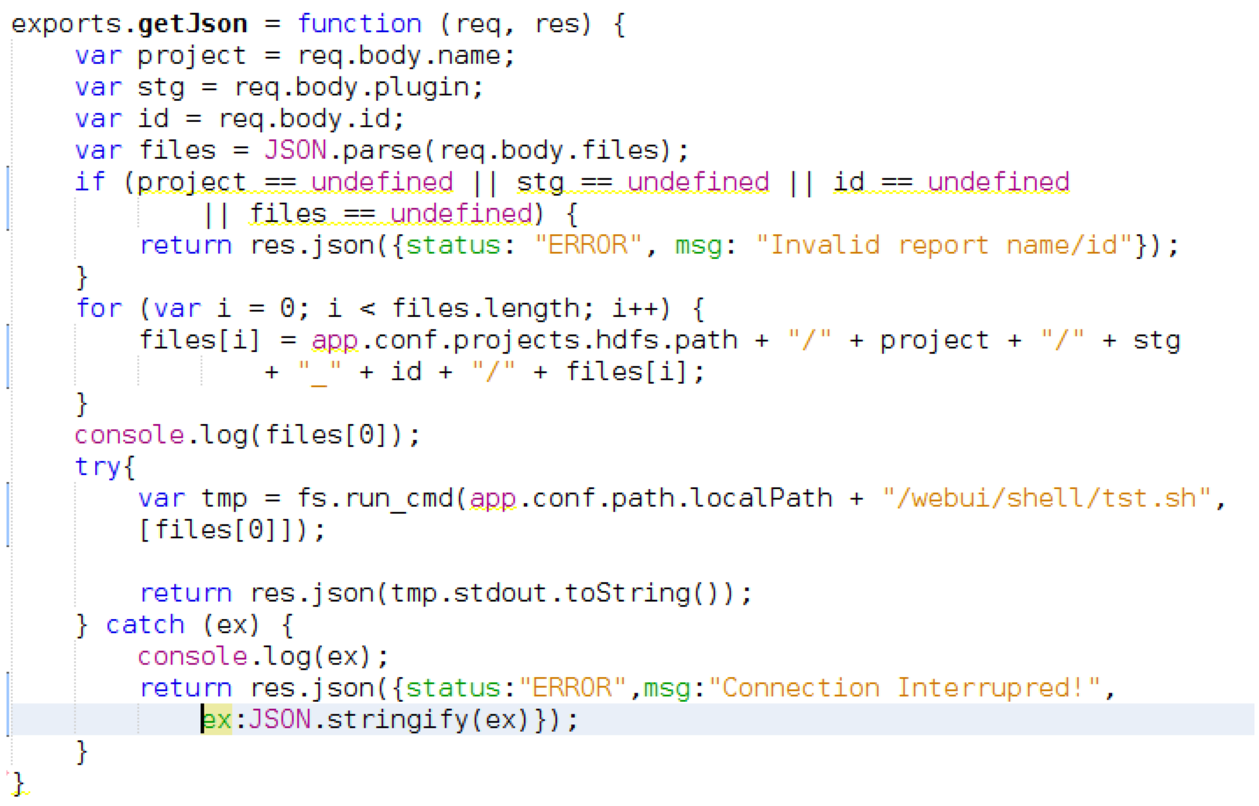
```
exports.compileProject = function (req, res) {
    if (req.body.name == undefined || req.body.name == null) {
        return res.status(400).send({status: "ERROR", msg:
                    "Invalid Project Name/ Project not found!"});
    }
    try {
        var port = global.compilerSocketPort - 1;
        var checkPort, output = "NA";
        while (output != undefined && output != "") {
            port = port + 1;
            checkPort = new fs.run_cmd("lsof", ["-i:" + port]);
            output = checkPort.stdout.toString();
        }
        console.log("sending output to" + port);
        //pass the project full path+name to compiler
        var compile = new fs.
                runCmd(app.conf.path.localPath + "/webui/shell/websocketd",
                ["--port=" + port, "java", "-jar",
                    app.conf.compiler.path + "/" + app.conf.compiler.executable,
                    "VERBOSE",
                    "--variable:pluginsPath=" + app.conf.plugins.path,
                    "--variable:coreLibPath=" + app.conf.library.core.path,
                    "COMPILE",
                    app.conf.projects.localPath + "/" + req.body.name,
                    app.conf.projects.localPath + "/" + req.body.name,
                ], function (o, e) {
            console.log(o + "\n" + e);
        });

        var pid = fs.run_cmd("echo", ["$!"]);
        return res.json({status: "OK", msg: "Command Executed!",
            data: {port: port, pid: pid.stdout.toString()}});

    } catch (e) {
        console.log(e);
        return res.status(500).send({status: "ERROR", msg: "Server Error!"});
```

Fig. 27 Compilation Subroutine

Once the user has created the topology and clicks on the “Compile” button, the compilation subroutine (see Fig. 27) is called on the server side. The subroutine takes the project path name from the request object sent from client side and executes the java compilation shell command on the source code generated from the topology JSON file. The output is an executable JAR file which is a deployable Apache Spark job. The “*Run*” button is used to send the run request to the server side, which is processed at the server side by executing another shell command that runs Apache Spark's tool to actually deploy the jar to the execution engine.

4. Reports sub-module

   A report is generated whenever there is an Apache Spark job being executed and the job topology contains at least one "*ReportStage*" implementation plugin. The live report is represented to the end user on the reports viewable area and is updated every second by fetching the report data from the server side. As shown in the Fig. 30, the figure shows a live report generated with the help of a report plugin implementation. This particular report plugin generates a line chart, as shown in the Fig. 30, and outputs number of filtered tweets generated per second, based on a particular filter word specified by the end user in the prior filter plugin in the topology. On the server side of the "*webui*" module, the reports sub-module is responsible to fetch the report data from Apache Hadoop HDFS node and give it to the client side. As shown in the Fig. 29, the subroutine uses the request object sent from client side, extracts the project-id, stage-name and file paths that hold the report data. Then, the subroutine, iterates over the file names and runs a shell script named "*tst.sh*" that actually executes the "*hdfs*" command to download the report data internally. Each report file's data is returned back to the client through a state-full connection.



```
exports.getJson = function (req, res) {
    var project = req.body.name;
    var stg = req.body.plugin;
    var id = req.body.id;
    var files = JSON.parse(req.body.files);
    if (project == undefined || stg == undefined || id == undefined
            || files == undefined) {
        return res.json({status: "ERROR", msg: "Invalid report name/id"});
    }
    for (var i = 0; i < files.length; i++) {
        files[i] = app.conf.projects.hdfs.path + "/" + project + "/" + stg
                + "_" + id + "/" + files[i];
    }
    console.log(files[0]);
    try{
        var tmp = fs.run_cmd(app.conf.path.localPath + "/webui/shell/tst.sh",
        [files[0]]);

        return res.json(tmp.stdout.toString());
    } catch (ex) {
        console.log(ex);
        return res.json({status:"ERROR",msg:"Connection Interrupred!",
            ex:JSON.stringify(ex)});
    }
}
```

Fig. 28 Subroutine to fetch JSON report from HDFS node

4.3 Simulation Procedure

To run the Stream Analyzer software, follow the procedure below:

1. Download and install Apache Spark 2.0 and Apache Hadoop 2.7. These are the execution engines on which the software is based. These both engines need to be up and running before starting up the “*webui*” of Stream Analyzer project.

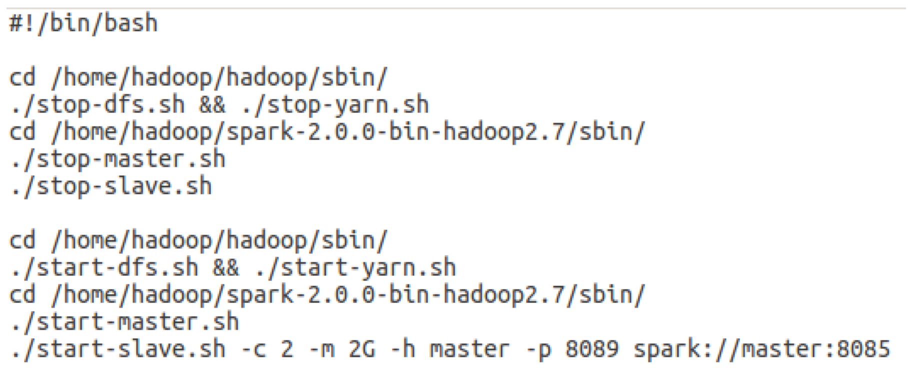

```
#!/bin/bash

cd /home/hadoop/hadoop/sbin/
./stop-dfs.sh && ./stop-yarn.sh
cd /home/hadoop/spark-2.0.0-bin-hadoop2.7/sbin/
./stop-master.sh
./stop-slave.sh

cd /home/hadoop/hadoop/sbin/
./start-dfs.sh && ./start-yarn.sh
cd /home/hadoop/spark-2.0.0-bin-hadoop2.7/sbin/
./start-master.sh
./start-slave.sh -c 2 -m 2G -h master -p 8089 spark://master:8085
```

Fig. 29 "start_services.sh" Script to start execution engines.

As shown in the Fig. 29, the script named "*start_services.sh*" is executed to stop any running execution engines and start them again.

2. Go to the Stream Analyzer's root installation directory, and run following commands:

   1. $cd webui – changes the current working directory to "*webui*" folder

   2. $npm start – starts the main web GUI.

3. After the "*webui*" is started, the GUI backend server starts listening to any request that comes from the client side. Now we need to open the actual web GUI. Open any modern web browser, for example Google's Chrome, Mozilla's Firefox or Internet Explorer (9+). Type down the address of the host on which the Stream Analyzer is running (probably your localhost), in the location bar of the web browser.

   URL : *http://localhost(or the host's ip address):8000*

4. Now the GUI should be loaded on the web frontend. It will ask to either enter a user email-id and password or register for a new one. Do the needful.

5. Once logged-in to the GUI, a dashboard screen is presented which shows the status of all the previously created projects and the execution engines as well.

6. Now, visit the “*Projects -> Create*” menu present on the top and the topology design area is presented. End user can choose any component available on the left pane to drag into the design area and connect to any other component for data in-out flow. The user also needs to configure the configuration parameters for each component on the design area.
7. After designing the topology, user can save the project by clicking the “*Save*” button after providing a name to the project in the project name field present on the top left of the topology design screen. Once saved, the project needs to be compiled by clicking the “*Compile*” button (right next to the “*Save*” button). The compilation output is shown on the output terminal pane at the bottom of the screen,
8. Once compiled, click on the “*Run*” button. The link to view the final live report is shown in the terminal pane. Click the link to visit the live report page that shows live updates of the generated report for the topology/job under execution.

4.4. Results

A live report is the data visualization of the expected output of the running job. We ran the Twitter Stream Job in two ways:

1. First we created a basic program using the Spark core library. We ran the program on to the spark cluster and achieved average number of tweets processed per second was ~ 50 to 60 with 4 workers (2GB RAM and 1 CPU each).

2. Second time we developed the job using Stream Analyzer as shown in Figure 5. On running the job through stream analyzer system, we saw negligible lag while processing stream of tweets. We found it to be around 45 to 60 tweets per second with peak at 100 tweets/second when deployed to the same spark cluster i.e. 4 workers, 2GB ram and 1 CPU each. The live report can be seen in Figure 30.

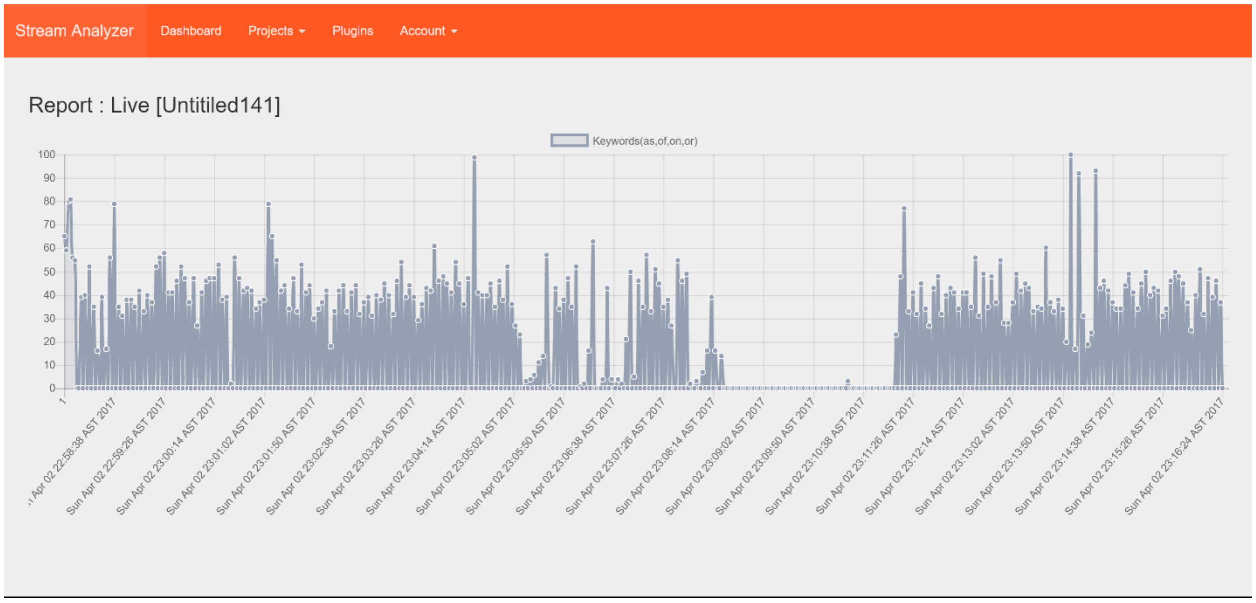

Stream Analyzer
Dashboard
Projects
Plugins
Account
Report : Live [Untitiled141]
Keywords(as,of,on,or)
100
90
80
70
60
50
40
30
20
10
0
Sun Apr 02 22:58:38 AST 2017
Sun Apr 02 22:59:26 AST 2017
Sun Apr 02 23:00:14 AST 2017
Sun Apr 02 23:01:02 AST 2017
Sun Apr 02 23:01:50 AST 2017
Sun Apr 02 23:02:38 AST 2017
Sun Apr 02 23:03:26 AST 2017
Sun Apr 02 23:04:14 AST 2017
Sun Apr 02 23:05:02 AST 2017
Sun Apr 02 23:05:50 AST 2017
Sun Apr 02 23:06:38 AST 2017
Sun Apr 02 23:07:26 AST 2017
Sun Apr 02 23:08:14 AST 2017
Sun Apr 02 23:09:02 AST 2017
Sun Apr 02 23:09:50 AST 2017
Sun Apr 02 23:10:38 AST 2017
Sun Apr 02 23:11:26 AST 2017
Sun Apr 02 23:12:14 AST 2017
Sun Apr 02 23:13:02 AST 2017
Sun Apr 02 23:13:50 AST 2017
Sun Apr 02 23:14:38 AST 2017
Sun Apr 02 23:15:26 AST 2017
Sun Apr 02 23:16:24 AST 2017

Fig. 30 Report (Line Chart for Twitter Stream Demo Topology)

We can see that the system performs almost equal to what a hand coded job would perform like on a spark cluster. There is not much overhead code added to the spark job when designed and compiled through the Stream Analyzer system.

5. CONCLUSION & FUTURE WORK

5.1 Conclusion

1. The main objective of developing a web based GUI interface for distributed processing application development was achieved. We developed a graphical development interface (GDI) which can be used by end users to design and deploy distributed processing job topologies by simple means of dragging and dropping the required parts of the topology on design area and connecting them for in and out flow of data to be processed.

2. We developed a cross platform software that can easily be run on any major operating system. The only major requirement for it to run is presence of open sourced development frameworks named “NodeJS” and “Java SDK”.

3. We achieved the objective of developing an easily accessible GUI which can run any major web browser. End users can easily access the GUI by visiting the GUI’s link on any web browser and login to the system to access the topology design area.

4. As, the end users can create their own accounts and profiles, in this way, the system is providing a workspace to each user, in isolated way, so as to let the user store his/her projects and topologies separate from other users.

5. We achieved the aim of developing an open sourced framework by making the system plugins based. End users, topology developers or any other experienced programmer can

develop their own implementations of map reduce actions into plugins and freely get the plugins developed by other users. They can also distribute the plugins they create, online.

5.2 Limitations and Future Scope

Following are the limitations that we found till now:

1. The system is based out of apache spark engine. Integration of another engine (like Flink) is a big hurdle.

2. For each specific kind of big data problem to solve, there is a need for existence of relevant plugins/components to design the topology, already on the web. This means that for each specific problem to be solved, especially complex one, the plugins need to be hand coded first and distributed online to let other's solve similar problems easily.

3. Another big limitation of this system till date is that it supports only one stream of data per topology. Apache spark doesn't allow multiple streams implementation in single job. However, to solve this problem, in future we can integrate Apache Flink as a processing engine in addition to Apache Spark.

References

1. Moore’s Law

   https://en.wikipedia.org/wiki/Moore%27s_law

2. Apache Hadoop Project

   http://hadoop.apache.org/

3. Apache Spark Project

   https://spark.apache.org/

4. A Comprehensive View of Hadoop Research - A Systematic Literature Review

   Ivanilton Polatoa, Reginaldo Re, Alfredo Goldman , Fabio Kona, Department of Computer Science, University of S˜ao Paulo, S˜ao Paulo, Brazil bDepartment of Computer Science, Federal University of Technology - Paran, Campo Mour ˜ao, Brazil,

   https://www.ime.usp.br/~kon/papers/Polato_JNCA_2014.pdf

5. Spark: Cluster Computing with Working Sets, Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica University of California, Berkeley,

   http://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf

6. The Hadoop Distributed File System Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler Yahoo! Sunnyvale, California USA {Shv, Hairong, SRadia, Chansler}@Yahoo-Inc.com

   http://storageconference.us/2010/Papers/MSST/Shvachko.pdf

7. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,

Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica University of California, Berkeley

http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf

8. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica University of California, Berkeley

   http://people.csail.mit.edu/matei/papers/2012/hotcloud_spark_streaming.pdf

9. A Review Paper on Big Data and Hadoop Harshawardhan S. Bhosale1 , Prof. Devendra P. Gadekar2 1Department of Computer Engineering, JSPM's Imperial College of Engineering & Research, Wagholi, Pune Bhosale.harshawardhan186@gmail.com 2 Department of Computer Engineering, JSPM's Imperial College of Engineering & Research, Wagholi, Pune devendraagadekar84@gmail.com

   http://www.ijsrp.org/research-paper-1014/ijsrp-p34125.pdf

10. Talend Studio Project

    https://www.talend.com/

11. Pentaho Project

    http://www.pentaho.com/

12. IBM (a multinational company) article on Data Science.

    https://www.ibm.com/blogs/watson/2016/05/biggest-data-challenges-might-not-even-know/